Statistical runtime verification of agent-based simulations

Herd, Benjamin

Awarding institution:
King's College London

KING'S COLLEGE LONDON

Statistical runtime verification of agent-based simulations

Benjamin Herd

A thesis submitted in partial fulfillment for the
degree of Doctor of Philosophy

in the
Department of Informatics
Faculty of Natural & Mathematical Sciences

February 2015

Abstract

As a consequence of the growing adoption of *agent-based simulations* as decision making tools in various (potentially also critical) areas, questions of veracity and validity become increasingly important. In general software and hardware development, *formal verification* – particularly *model checking* – has been applied successfully to a wide range of problems; due to their immense complexity, however, agent-based simulations lend themselves to conventional formal verification only in very simple cases and at a disproportionately high cost.

The purpose of this work is to address this problem and present a *statistical runtime verification approach* which focusses on the analysis of the temporal behaviour of large-scale probabilistic agent-based simulations. The approach is tailored to the particular mix of characteristics that agent-based simulations typically exhibit: large populations, randomness, heterogeneity, temporal boundedness and the existence of multiple observational levels. It combines the ideas of *runtime verification* and *statistical model checking* and allows for the temporal verification of simulations with hundreds or thousands of constituents and probabilistic state transitions. Instead of requiring a formal model, verification is performed upon traces of the original simulation obtained through repeated execution. Properties are checked on-the-fly, i.e. during the execution of the simulation, which is achieved by interleaving simulation and verification. Evaluation is lazy, i.e. a simulation step is performed only if the property has not already been satisfied or refuted. This reduces the amount of simulation to a minimum and restricts state space exploration to the smallest fragment necessary for finding a definite answer to the given property. Verification results are approximate, but the precision is clearly quantifiable and adjustable by varying the number of simulation runs.

To my family

Acknowledgements

I would like to express my sincere gratitude to

- Dr Simon Miles, Professor Peter McBurney and Professor Michael Luck for their excellent supervision, for giving me the freedom to pursue my own research interests and for great guidance in all aspects of academic life. Combining work and research would not have been possible without their support, encouragement and understanding of my professional circumstances.
- My examiners Professor Alessio Lomuscio and Dr Steve Phelps for their encouraging and constructive feedback and for providing thoughtful and detailed comments about this (fairly long) piece of work.
- The team of Sandtable Ltd in London, particularly Mr Andrew Skates, for always being supportive of my research ambitions, for great discussions, and for giving me the opportunity to evaluate my ideas in a real-world environment.
- My parents for always standing by me, believing in me and backing all of my decisions, no matter how crazy they were. I wouldn't be even close to where I am without your love and unconditional support.
- Julia for her patience and understanding, her efforts to keep my motivation up and for endless discussions about agents and state spaces. Thank you very much! I owe you several holidays.

Contents

List of Algorithms

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The emergence of agent-based modelling

Agent-based modelling or *agent-based simulation*[1] is rapidly emerging as a popular paradigm for the simulation of complex systems that exhibit non-linear and emergent behaviour. It uses populations of interacting, autonomous and often intelligent agents to model and simulate various phenomena that arise from the dynamics of the underlying complex systems. Influenced by (distributed) artificial intelligence, complexity science and computer simulation, agent-based simulation is being applied successfully to an ever-increasing number of real-world problems and could in many areas show advantages over traditional numerical and analytical approaches; as a consequence, it is often being employed as a decision support tool for policy making and analysis. However, although social science has been its traditional domain, agent-based simulation is also increasingly being used for the analysis of complex technical, in many cases also safety-critical, systems in areas such as avionics [38], airport performance modelling [39], or the design and analysis of robot and UAV swarms [179, 236].

Despite its increasing adoption, however, many people are still very sceptical of the benefits of agent-based modelling as opposed to more traditional, better understood techniques such as differential equations or econometrics. There are at least three main reasons for this scepticism, which we discuss in turn. First, due to their high level of complexity, agent-based simulations are notoriously difficult to

[1] In this work, we use the terms *agent-based modelling* and *agent-based simulation* interchangeably.

engineer, to understand and to control. The high level of complexity arises from a combination of characteristics which may not be problematic on their own, yet certainly are in their combination. Some of these characteristics are listed below.

- Agent-based simulations have many constituents (often thousands of agents) which need to be engineered, coordinated and tested.
- The behaviour of agent-based simulations can be observed on at least three different levels, *micro* (individual), *macro* (global) and *meso* (intermediate, interaction-centric).
- As opposed to *microsimulations* where agents are represented explicitly but independent from each other and often as simple probabilistic processes [107], agent-based simulations typically exhibit a high level of interaction among agents as well as significant heterogeneity and complexity with respect to the individual agent behaviour.
- One of the strengths of agent-based simulations is their capability to exhibit emergent macro-level behaviour which is by definition not (or at least not easily) reducible to the behaviour of the individual components; these simulations thus invariably contain an element of surprise which makes their behaviour hard or even impossible to predict and, conversely, makes it hard to engineer a system deliberately with a certain desired macro behaviour in mind.
- Most agent-based simulations contain a significant amount of randomness, both as part of the individual agent's decision making process and in the environment's dynamics; this requires the execution of a large number of simulation runs in order to average over stochastic variance.
- Especially in the area of social simulation, the behaviour of individual agents often involves fairly complex, domain-specific calculations as well as the integration of external data (often resulting from historic observations).

A second, more social, reason for the scepticism associated with agent-based modelling is the fact that the technique is still relatively new. The first larger-scale agent-based model in the social sciences was developed and published by Epstein in 1996 [89], Gilbert and Troitzsch published the first book on social simulation in 1999 [109], and the *Journal of Artificial Societies and Social Simulation* — the first journal dedicated to the new field — was established in 1999. Due to its young age, agent-based modelling competes with established, well understood techniques such as econometrics [52] or system dynamics [37] and needs to prove itself robust and reliable before gaining wider acceptance. The

emergence of agent-based modelling is a typical example of a paradigm shift, a shift from the focus on large populations of mostly unconnected individuals, from normal distributions and the law of large numbers to complex systems whose constituents are essentially connected, to power law distributions and emergent phenomena. This shift will take time and the only way for agent-based modelling to become accepted is to show its usefulness for the solution of hard real-world problems which conventional techniques are not or insufficiently capable of solving.

A third reason for the widespread scepticism about the usefulness of agent-based simulation is that the resulting models are often seen as black boxes whose internal dynamics are insufficiently well understood [234, 154]. The complexity issues mentioned above certainly have a significant impact on this view; when constructing models, users are often confronted with an overwhelming amount of information which needs to be visualised, processed, and made sense of.

As mentioned above, agent-based simulations are increasingly being used as decision making tool for policy analysis and design as well as for the simulation of potentially safety-critical systems. In that context, and similar to other software systems, *correctness* plays a central role and questions of quality assurance become increasingly important. Dedicated methods and tools for the verification and formal analysis of agent-based simulations, however, are still largely missing. The purpose of this research project is to address this problem. The next section gives a more detailed motivation for the project by elaborating on some of the typical issues in correctness checking in the context of agent-based simulation.

1.2 Research problem

As a consequence of the aforementioned facts — the rapid emergence of agent-based modelling, its widespread adoption as decision making tools and, on the other hand, the widespread scepticism associated with it — it is crucial to find ways to *increase trust* by means of reliable and continuous model verification. The purpose of this work is to address this problem and develop a technique which can help to increase confidence in the simulation results by automating model exploration and qualitative and quantitative analysis. Before we describe our research ideas, however, it is useful to briefly elaborate on the idea of increasing trust and to discuss potential ways in which this can be achieved.

We believe that, precisely due to the complexity issues described above, trust in agent-based models can only be increased by increasing their transparency, i.e. by revealing as many of their internal dynamics as

possible. We argue that this can only be achieved through continuous quality assurance, by continuously *monitoring* the system and detecting problems and inconsistencies as early as possible. In agent-based modelling, it is, for example, common to construct a model incrementally, starting with a very simple representation of the problem under study and gradually increasing the complexity of the model by adding new features [167]. However, every addition of a new feature introduces complexity and may negatively impact the understandability of the model's internal mechanisms. Due to nonlinear dynamics and feedback mechanisms, even small changes can have large consequences where one would have least expected them; on the other hand, desired effects may be cancelled out entirely. The modelling process is typically highly iterative in nature; insights gained from one version of the model determine the requirements for the next version. As a consequence, problems are likely to multiply if they are not detected early enough. It is therefore crucial to detect problems as early as possible, according to the principle, 'fail early, fail fast, fail often' [222].

The purpose of this section is to motivate the research project in more detail. We start with a simple but realistic example model and typical correctness questions. We then discuss why existing verification approaches are, in general, not sufficient to address the particular mix of characteristics which can be found in many agent-based simulations. And finally, we show that three particular characteristics — (i) a large number of constituents, (ii) different observational levels and (iii) randomness — impose additional requirements on the formulation of useful correctness properties which are not satisfied by current approaches.

1.2.1 Motivating example

Consider a simple disease transmission model which aims to represent the propagation of, for example, a virus in a population of 100 agents. For simplicity, we assume that the 'contact network', i.e. the topology based on which agents interact, is given in advance (see Figure 1.1). The mechanisms based on which agents pass on infections can be manifold and need not concern us further here. In a concrete model, they will be typically based on a combination of each agent's internal state and the state of the agent's neighbourhood. Transmission models can serve as a metaphor for a wide variety of real-world phenomena such as the spreading of diseases, the propagation of information (e.g. rumours), the diffusion of viruses in computer networks or the adoption of technologies within a society.

As a modeller, the major focus of most efforts is to understand how such a system behaves over time. We may, for example, want to know whether, starting with a single infected agent, it is possible to eventually

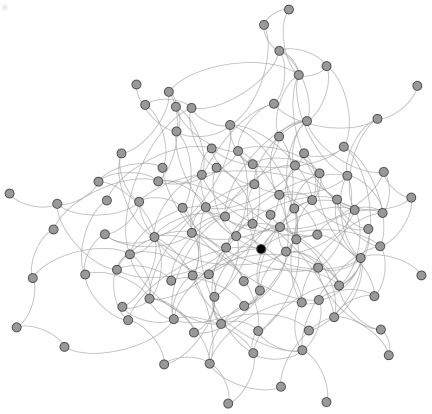


FIGURE 1.1: An example network of 100 agents

reach a state in which all (or a significant proportion of) agents are infected. We may also want to know whether the choice of an initially infected agent makes a significant difference to the overall probability that the entire population (or a certain proportion thereof) becomes eventually infected. Or, as a third example, suppose that agents have an additional attribute $gender \in \{m, f\}$ which also has an influence on an agent's probability of becoming infected; in this case, we may be interested in whether the overall probability of the male fragment of the population becoming infected is smaller, equal to or larger than that of the female fragment.

Heterogeneous systems like agent-based simulations are open to a wide range of correctness questions. But even for simple systems such as the one given above, the temporal dynamics may become extraordinarily complex. If we assume that each agent can be in one of just two individual states (e.g. $infected$ and $healthy$), then the number of possible global states already amounts to $2^{100} \approx 1.27 \cdot 10^{30}$. In the presence of (i) concurrency, i.e. variations in the order of agents updating their state, and (ii) randomness with respect to the individual agents' decision making process, the chance of a significant fraction of this theoretical state being actually reachable is fairly high. Answering questions about the global behaviour of such systems accurately thus amounts to searching through a vast space which becomes exceptionally hard or even infeasible.

Answering correctness questions like the ones above is the purpose of verification techniques such as, for example, *temporal logic model checking*. In the next section, we therefore briefly mention the particular strengths and weaknesses of existing verification approaches and their applicability to the verification of agent-based simulations as motivated above.

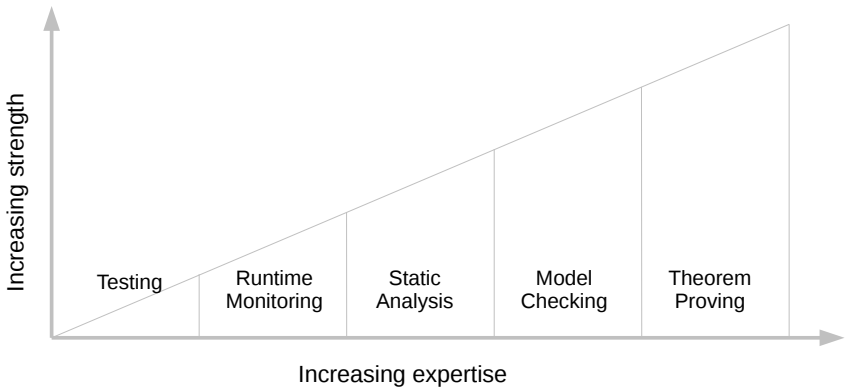


FIGURE 1.2: Classification of verification techniques (adapted from [183])

1.2.2 Static and dynamic verification

Existing verification techniques can be classified according to their strength and the expertise needed to carry them out (see Figure 1.2) [183]. Furthermore, they can be subdivided into *static* and *dynamic* approaches. Instead of executing and analysing the program at runtime, static approaches use formal methods which are applied to the source code or an abstract model of the program in order to assess the correctness. Typical examples of static verification techniques are *static code analysis*, *theorem proving* and *model checking*. Due to their formal nature, static verification techniques are able to produce solid and trustworthy results. On the other hand, their design and execution requires a comparatively high level of expertise and, due to their exhaustive analysis of the underlying system, static techniques often face the problem of exponential explosion. This is particularly critical for concurrent systems in which actions may be executed in arbitrary order. In order to avoid contingency effects, agents in an agent-based simulation are often updated in random order which has a similar effect on the nature of the underlying state space as true parallel execution. Even if not properly parallelised and executed on parallel or distributed hardware, agent-based simulations are thus often inherently concurrent in nature.

In contrast to static verification, dynamic verification approaches are applied at runtime and analyse a program's *execution trace* in order to verify given properties. Typical examples of dynamic approaches are *run-time monitoring* and *testing*. Due to their focus on runtime execution, they avoid most of the

complexity problems which are inherent to static techniques; due to their focus on the original system, they also provide more *immediacy* than static approaches which operate upon a formal (and often simplified) representation of the system under consideration. Their lightweight nature and the comparatively flat learning curve are further aspects which contribute to the success of dynamic software testing over formal methods — especially in industry.

However, the convenience comes at the expense of verification strength. Since dynamic approaches take into account the current execution trace only, they are inherently limited to a particular control flow; this allows them to show the presence of errors, but not to prove their absence. Achieving good coverage by executing the program (or parts of it) several times in order to reach as many different control paths as possible is therefore essential for the strength of the verification process. Complexity also poses problems — especially in the area of monitoring. The amount of information flowing through a system can be overwhelming for both developers and users. This is particularly critical if large-scale systems with hundreds or thousands of constituents and potentially concurrent behaviour need to be monitored. Deciding which information to output (and doing so correctly) is critical but can be highly non-trivial.

Runtime verification refers to the idea of *observing* the execution of a program at runtime and performing correctness checks 'on-the-fly' [155]. A central component of any runtime verification approach is a *monitoring process* which observes a trace against the background of given correctness criteria (e.g. formulated in linear temporal logic) and reports for each state in the trace whether the given formula has already been satisfied or violated. As opposed to full model checking and similar to testing, runtime verification operates upon the running system and checks its execution traces against a set of verification properties; as opposed to testing in which the correctness criteria are represented as *test suites*, i.e. finite sets of input-output sequences, properties in runtime verification are often formulated in temporal logic. Runtime verification provides a convenient balance between rigorous and strong but complex formal verification and efficient but significantly weaker conventional software testing [155]. It can thus be seen as a more lightweight alternative for systems which are unamenable to formal verification.

Over the years, a number of static and dynamic verification approaches for general, both non-probabilistic and probabilistic, multiagent systems have been presented (a brief overview is given in Section 2.5). Due to the high complexity of these systems, combinatorial explosion is a critical problem — particularly in the area of static verification. In order to address this problem, a wide range of abstraction and reduction approaches have been developed. Despite the usefulness of abstraction and reduction, however, there is always a tension between the size of the finite-state model used for verification and its representational accuracy; in order for the verification process to produce useful results, the model has to capture the

properties of the real-world system to a sufficient degree. An oversimplified abstraction, albeit fitting well into memory, might omit too much information about the target system and thus fail to represent it with a sufficient level of accuracy. This problem is largely ignored by current approaches which mostly concentrate on the verification of systems with a comparatively small number of components. Their main focus of interest is on the micro level, i.e. on the correctness of agent internals such as goals, plans and actions, or on the interaction among a small group of agents. A particular emphasis is put on agent-specific issues such as the consistency of BDI-based reasoning, the compliance of agents with a given communication protocol or the correctness of desired epistemic properties. Due to their agent-centric focus as opposed to a more global view on large agent populations, their adaption to the simulation domain is not straightforward. The verification of global properties of large-scale populations including stochastic behaviour has been addressed only recently [141].

Although existing approaches in the multiagent domain are capable of verifying certain aspects which are also relevant for agent-based simulations, approaches which specifically target their particular mix of characteristics (some of which have been mentioned in Section 1.1) are still missing. Some of the characteristics such as the global focus of interest and the high number of agents make the global behaviour of agent-based simulations particularly irreducible to their constituents and hence computationally difficult or even impossible to analyse using existing techniques. In the presence of large populations, issues of granularity and quantification arise (see Section 1.2.3). In order to understand the behaviour on multiple observational levels, a distinction between the behaviour of agents, of groups of agents or of the entire population becomes relevant. Other characteristics such as the high degree of randomness demand for qualitative and quantitative analysis, the verification of *expected* behaviour or the efficient verification of state transition probabilities without having to represent the system exhaustively (see Section 1.2.4). And finally, the complex logic of many simulation models severely complicates their formulation in a dedicated model description language required by most formal verification tools.

Furthermore, in the absence of a dedicated specification language, the formulation of complex, temporal testing criteria can be challenging; it is not immediately clear how, for example, a requirement like "*every time X happens, Y will also eventually happen at a later point in time*" can be formulated in a conventional testing environment elegantly without having to write complex testing code which is itself error-prone. In statistics, related questions can be answered by measuring the *internal correlation* or *autocorrelation*, i.e. the internal association between observations, of a given time series[2]. In the area of formal verification, temporal logic offers a convenient and purely declarative formalism for the

[2]The correspondence between temporal logic model checking and time series analysis is explored further in Section 7.4.

formulation of complex temporal properties. In the presence of large agent populations with different types of behaviours emerging on different observational levels, however, questions of *granularity* arise. This is illustrated in the following section.

1.2.3 Granularity

In the software and hardware verification literature, basic verification properties are typically classified as *reachability*, *safety* and *liveness properties*. A reachability property stipulates that, given an initial state, a certain goal state (or set of goal states) is reachable; a safety property describes that some bad behaviour will *never* happen; a liveness property stipulates that some good behaviour will *always* happen. A wide range of typical requirements for concurrent systems such as deadlock freedom, livelock freedom or eventual consistency can be formulated as reachability, safety and liveness properties.

Although the idea of reachability, safety, and liveness checking is typically associated with the verification of 'real' systems rather than with the analysis of simulation models, we argue that it forms an equally useful basis for correctness checking also in the latter case. For example, consider again the transmission model introduced in Section 1.2.1. It is easy to imagine the usefulness of the following example properties about the behaviour of individual agents in the model:

- It is possible for an agent to eventually become infected (reachability)
- An agent must never become infected (safety)
- When infected, an agent should always be able to recover (liveness)[3]

Given the large variety of analyses that a modeller may want to use an agent-based model for, the properties above may seem rather trivial. However, in order to use a simulation model as a basis for more complex experimentation, it is crucial that its basic internal mechanisms — such as the ones above — are working correctly. As described in more detail in Section 2.2.3, the focus of this work is on *internal validation*, i.e. on the correctness of the model's internal, causal mechanisms. In the presence of complex interactions between agents and the environment, even simple temporal behaviours like the ones above may be hard to impossible to predict from the individual rules built into the model. Being able to formulate basic assumptions about the model and check them automatically is therefore

[3] It is important to note that liveness properties are typically formulated over infinite traces. In a temporally bounded scenario (such as in simulation), liveness properties are thus only of limited usefulness. This is reflected in the semantic issues of temporal operators when being evaluated on finite traces described in Section 5.2.2.

an important prerequisite for robust experimentation. In order to 'open up' the blackbox of a complex simulation model, questions of reachability, safety, and liveness can be very helpful. However, it is also important to note that simple reachability, safety, and liveness checking is generally not sufficient for ascertaining internal validity. As we show in the remainder of this work (especially in Chapter 7), more complex properties about correlations of events and conditional and causal relationships are also required.

Analogous to the individual examples given above, it is equally easy to imagine similar properties about the behaviour of the entire population by, for example, replacing 'the agent' with 'the population', as shown below:

- It is possible for the whole population to eventually become infected (reachability)
- It is never possible for the population to become infected (safety)
- When infected, the population should always be able to recover (liveness)

When considering (possibly large) groups of agents, however, questions of quantification arise. Consider, for example, the third property from the list above. What does an 'infected population' mean? Are all agents required to be infected in order for the population to be deemed 'infected'? Or most of them? If the latter, what does 'most' mean? A language for formulating statements about groups of entities clearly needs to allow for the formulation of quantified statements of that sort. Classical logic provides existential and universal quantifiers which allow for a certain level of granularity and can be easily integrated into temporal logic; what they do not allow for, however, is *numeric* or *relative* quantification. This is required to formulate statements such as *"at least 100 agents are infected"* or *"80% of the agents are infected"*.

Apart from quantification, certain properties may only be required to hold for a particular group of agents which forms a subgroup of the entire population. For example, despite representing a much larger population in a disease transmission model, we may only be interested to formulate statements about the health behaviour of agents which are twenty to thirty years old. In conventional temporal logic, this is not straightforward. Another related and distinctive feature of complex systems is that they comprise multiple levels upon which behaviour can be observed; this is related with the phenomenon of emergence. Epstein, for example, showed that, although agents in a simple grid environment were only allowed to move up, down, left and right, the group as a whole moved diagonally [89]. It was therefore possible to observe behaviour on the group level which was not explicitly built into the agents' individual

rules; in order to detect and measure it, it would thus be necessary to formulate according properties about the group *as a whole*.

Agent-based simulations are meant to represent a real-world system with a sufficient level of accuracy. As a consequence, correctness checking should, in general, neither be limited to the micro nor to the macro level. As part of the verification and validation process, it is beneficial (and, in fact, also necessary) to perform correctness checking on all observational levels and align the behaviour of individual agents, groups of agents and the whole population with the behaviour of their real-world counterparts.

The discussion so far can be summarised as follows. The usage of existing verification techniques for the verification of agent-based models is, in theory, possible, albeit for a very limited class of agent-based models and associated correctness criteria. In the case of static verification (i.e. temporal logic-based model checking) complexity issues arise; the population is required to be either entirely homogeneous or sufficiently small, as are the internals of the individual agents. Furthermore, formulae are required to be sufficiently simple, i.e. either existentially or universally quantified. Dynamic verification (i.e. testing and runtime monitoring) avoids the complexity issues of static approaches at the expense of verification strength. Furthermore, in the absence of a dedicated specification language, correctness properties cannot be formulated in a declarative, elegant programming language-independent way.

Randomness plays a central role in agent-based modelling. As described in the next section, this poses additional problems which existing techniques are only insufficiently capable of addressing.

1.2.4 Randomness

In a deterministic scenario, verification questions can be given clear *yes/no* answers: a property is either true or false. As soon as the underlying system exhibits randomness, however, this need no longer be the case; here, verification amounts to determining the *probability* with which a given property holds rather than its absolute truth. Imagine, for example, the behaviour of an agent in the transmission example introduced above to be driven by stochastic rules; in this case, contact with an infected agent may not *necessarily* cause an 'infection', it may only do so with a certain probability. Verification needs to take randomness into account and allow for the evaluation of statements such as "*the probability of an agent eventually becoming infected is less than* p".

Several existing model checking approaches support the verification of probabilistic systems; the combination of large populations of agents and randomness, however, poses additional problems which are

not soluble with existing techniques. In addition to formulating quantified statements about groups of agents, we may also want to formulate statements about the *expected behaviour* of an agent. It is, for example, common to keep agents within a certain group *homogeneous* with respect to their behavioural protocol but *heterogeneous* with respect to their parametrisation. This means that, despite all agents following the same behavioural protocol, their behaviour at runtime differs due to (i) different attribute values, (ii) random variance and (iii) differences in local influences. Fluctuations in behaviour are desired and help to make the model realistic, yet they also make the dynamics hard to understand and to test. Among all the fluctuation, it is thus often important to ensure the correctness of the *average* or *expected behaviour*. We may, for example, stipulate that, *on average*, an adolescent agent in the population should only have a probability of 5% to become infected with a particular disease in the course of its simulated lifetime. This property is not checkable in a conventional model checking setting.

Assessing the expected behaviour of individual agents is closely related to assessing the compliance with their desired behavioural protocol. In order to understand that, it is useful to consider some typical characteristics of the model construction process. In the world of agent-based modelling, agents are often implemented such that they follow clear *if-then* rules; as opposed to other types of agent-based systems which rely heavily on the notion of deliberative agents, it is not uncommon for agents in a simulation to be rather simple and reactive. In the simplest case, agents may just follow a predetermined stochastic protocol. It is common to start with such a simplistic system in order to study and understand the macro level behaviour of the system prior to increasing the agents' complexity. New features are then gradually added to the system. This may, for example, involve diversification of agent attributes, addition of behavioural rules as well as replacement of previously fixed transition probabilities with a more complex decision making process (possibly based on other attributes), to name but a few. Adding features to the computational design and replacing some simple features with more complex mechanisms, however, may disguise the originally intended behavioural protocol. Things get even more complicated when agents are placed in the computational environment and allowed to interact with each other. Despite their relative freedom to act in an 'autonomous' way, they should, of course, still comply with certain basic mechanisms assumed to be true in the domain under consideration. For example, in the disease transmission scenario, despite agents constantly interacting with different agents (both healthy and infected), it may certainly be unrealistic for them to switch constantly between being healthy and being infected. Although not originally intended and perhaps even accommodated in the behavioural protocol, the switching behaviour may be an artefact of the interplay between internal attributes, predetermined probabilities and interactions with other agents. It could even be the result of a numerical error in the code.

Checking an agent's compliance with its intended behavioural protocol amounts to verifying the correctness of its *state transitions*. In a probabilistic scenario, however, apart from checking whether each state is only succeeded by its legitimate successor states, the correctness of *transition probabilities* is also required. In probabilistic model checking, this problem has been addressed by using a branching time approach in which the temporal evolution of the system is represented formally as a probabilistic transition system which, starting in the initial state, forms a tree and which embraces the idea of *possible worlds*. Unfortunately, continuous branching also causes exponential growth in the underlying formal model and therefore conflicts with the scalability requirement posed above. In Section 7.5, we illustrate how in a linear time setting, i.e. despite the focus of analysis being on a linear individual simulation traces only, questions about transition probabilities can still be answered by interpreting the sample space underlying the model in different ways.

This work aims to address these problems and develop a technique tailored to the verification of agent-based simulations against the background of the aforementioned characteristics and requirements; the research goals are summarised in the following section.

1.3 Research goals

The overall goal of this thesis is to increase the rigour in agent-based model and simulation development by bringing ideas from formal software engineering into the modelling process. Specifically, we aim to

- clarify the notion of 'correctness' in the context of agent-based simulation and, consequently, identify different types of typical correctness properties against the background of simulation verification and validation
- develop a verification approach which is capable of answering fine-grained and temporal properties about agent-based simulations with their particular characteristics on multiple observational levels and in an adjustably accurate and timely manner
- develop a practical verification framework which incorporates the theoretical ideas and can be used to analyse a wide range of realistic agent-based simulations

The next section gives an overview of the work and the proposed approach.

1.4 Overview of the work

Considering the complexity of agent-based simulations on one hand as well as the advantages and disadvantages of both static and dynamic verification techniques on the other hand, we aim to combine the rigour and unambiguousness of model checking with the scalability, immediacy and efficiency of testing. The goal of this work is to develop a technique for the *approximate verification* of the *temporal* behaviour of *large-scale* and *probabilistic agent-based simulations*. In order to circumvent state space explosion, we aim for a *statistical approach* which evaluates properties upon a sufficiently large number of *simulation traces*. As opposed to other verification techniques in the more general field of multiagent systems [164], we restrict our focus to purely *temporal behaviour* and ignore higher-level notions such as, for example, knowledge, beliefs, intentions or strategies. We refer to our approach as *statistical runtime verification*. It is inspired by two existing ideas. The first idea is *statistical model checking* and, in particular, *approximate probabilistic model checking* which has been presented by Hérault and Lassaigne [117]; the second idea is that of *runtime verification* which dates back to the work of Chodrow, Kim, Lee and others in the 1990s [55, 137, 151].

Statistical (or approximate) model checking allows for the approximation of the expected probability of a formula to a tunable degree of accuracy by using a statistical sampling approach (see Section 2.4.3.3). This is particularly interesting for the verification of large-scale systems like agent-based simulations whose complexity makes them unamenable to exhaustive analysis. Due to their stochastic nature, agent-based simulations can be seen as special variants of Monte Carlo simulations and, as such, are naturally amenable to the idea of approximate model checking. The idea of runtime verification, on the other hand, allows for the verification of software systems 'on-the-fly', i.e. during their execution; by detecting satisfaction or refusal of a property as early as it happens, state space exploration can be reduced to a minimum. Due to the accuracy of approximate model checking being dependent upon the number of traces analysed, reducing the time spent for the evaluation of a single trace to a minimum is of critical importance. We therefore believe that the combination of both ideas serves as an ideal basis for the construction of a verification technique for potentially large-scale agent-based simulations.

1.4.1 Challenges

In alignment with typical challenges in the context of runtime verification, we focus on the following problems:

Property specification: We aim to develop a *property specification language* which allows for the formulation of complex properties about large-scale agent-based simulation traces in a *declarative* and *succinct* way.

Monitor generation: We aim to construct an *efficient monitor* which observes the behaviour of running simulations and reports evaluation results as soon as they can be detected. In doing so, we aim to put a particular focus on *optimality* with respect to the avoidance of unnecessary computation.

Advanced properties: Runtime verification is typically constrained to the verification of safety properties. In order to increase the usefulness of the developed approach, we aim to put a particular focus on the identification of properties which extend beyond pure safety statements. Examples for the advanced types of analyses that we aim to conduct are given further below.

Instrumentation: Instead of instrumenting existing model code, we aim to develop a *runtime verification framework* which *combines simulation and verification*. Individual simulation code can be embedded and checked with the help of a clearly defined *service provider interface (SPI)*.

Increasing coverage: Instead of focussing on individual traces, we aim to achieve good coverage of the underlying state space by means of *Monte Carlo simulation*, i.e. repeated execution of the underlying stochastic simulation, and *statistical analysis* in order to obtain accurate confidence intervals for the verification results.

Conventional runtime verification is typically focussed on the verification of pure safety properties. We aim for a technique which is additionally capable of performing the following types of analysis (illustrated with example questions related with the transmission example introduced above):

Probability analysis: What is the probability of certain *events*, i.e. temporal or atemporal phenomena, occurring in the course of the simulation? For example, *What is the probability of the majority of the population eventually becoming infected*?

Correlation analysis: Is there a correlation between two events A and B? For example, *Are male agents more likely to eventually become infected than female agents, i.e. is gender correlated to chance of infection*?

Conditional analysis: What is the conditional relationship between two events A and B? For example, *What is the probability of an agent's transitioning into state 'infected' after being in state 'healthy'*?

Causal analysis: Does a change to agent-internal or environmental conditions have a causal influence on the occurrence of certain events? For example, *Does increasing the probability of recovering increase the overall probability of the majority of the population eventually becoming infected*?

In order to allow for *multi-level correctness checking*, we further want to be able to apply the aforementioned types of analysis to *individual agents (micro)*, *groups of interacting agents (meso)* as well as *the entire population (macro)*, including appropriate quantification capabilities in the case of the latter two levels as described in Section 1.2.3 above. In order to account for random variance in large-scale stochastic simulations, we also want to be able to formulate statements about the *expected* or *average* behaviour of agents within a group.

Given the vast complexity of agent-based simulations, we cannot hope for a technique which is capable of exhaustively exploring their state spaces. We thus aim for an *approximate*, yet *quantifiably accurate* solution which allows a modeller to obtain approximate verification results together with clear statements about their precision.

1.4.2 Contributions

In alignment with the goals and challenges described above, our contributions are as follows:

- We develop a formalisation of agent-based simulations for the purpose of providing a semantic framework that the subsequent algorithms are to be defined upon (see Chapter 3).
- We give a formal definition of *events* as temporal or atemporal phenomena in a simulation context against the background of the underlying sampling space (see Chapter 3); we relate events with properties to be formulated and show that, by interpreting the sample space in different ways, different types of properties about arbitrary fragments of the simulated traces can be formulated and verified.
- After a semi-formal discussion of analysing individual simulation and the identification of requirements and basic data structures (see Chapter 4), we present *simLTL*, a LTL-based property specification language tailored to the formulation of correctness criteria about large-scale agent-based simulations (see Chapter 5). The language comprises syntactic constructs for advanced quantification, selection, access to agent attributes and aggregations thereof as well as simple

arithmetic relations. This allows for the formulation of typical correctness properties with different levels of granularity and on different observational levels as motivated in Section 1.2.3. In order to allow for the verification of simulations at runtime, the semantics of simLTL are defined over finite traces.

- We present a *statistical runtime verification* algorithm which allows for the verification of temporal properties of large-scale agent-based simulations formulated in simLTL in an approximate, yet adjustably accurate manner (see Chapter 6); the algorithm implements a *monitor* which reports the truth or falsity of a property as soon as it is clearly decidable. Furthermore, due to the runtime nature of the approach, simulation and verification are intertwined processes; as a consequence, only the fragment of the state space which is strictly necessary for answering a given property is actually explored. The probability of properties is determined through a combination of *repeated simulation* and *statistical analysis*.

- We describe the verification of *advanced temporal properties* which extend beyond pure safety checking (see Chapter 7). Those properties involve

  - correctness checking of transition probabilities
  - verification of the expected behaviour of individual agents or arbitrary groups of agents
  - analysis of correlations between events
  - analysis of conditional relationships between events
  - the detection of simple causal relationships between events

- We present MC^{2}MABS, a practical framework for the verification of correctness properties of agent-based simulations which implements the aforementioned concepts (see Chapter 8). The framework comprises a *Monte Carlo simulator* together with a *runtime model checker* which acts as a *monitor* for simLTL properties. The simulator allows for the formulation of the model logic in a high-level programming language; properties can thus be checked immediately on the underlying system rather than on an abstract formal model.

We further provide an evaluation of the verification approach in a larger case study in the area of swarm robotics, as described in Chapter 9.

In developing the approach, we put a particular focus on the following principles:

Immediacy: Traditional model checkers require a translation of the system logic into a dedicated modelling language. We believe that, in the area of agent-based simulations which often require complex agent-internal logic such as utility calculations, such a translation would be extremely limiting. Instead of providing a dedicated modelling language, we therefore propose a C++-based simulation framework. It allows for the formulation of the model logic in a general purpose programming language which facilitates the description of both simple state transition rules and arbitrarily complex high-level logic. Verification is thus performed on the original model rather than on an abstraction.

Expressiveness: The distributed nature of agent-based simulations and the potentially high level of heterogeneity within the population requires a finer-grained approach to property formulation than that provided by classical logic. For example, in addition to existential and universal quantification, statements about fractions and percentages of the population or subgroups thereof need to be expressible. Furthermore, due to the stochastic but also exploratory and predictive nature of agent-based simulations, advanced types of properties which make statements about the *expected behaviour* of individual agents, about the *behaviour of entire groups* as opposed to the behaviour of their members, about *correlated events* and *causal relationships*, are necessary for comprehensive quality assurance. simLTL allows for this by means of a differentiation between *agent-level* and *global-level formulae*, by *selection* and *quantification operators* as well as by a semantically realised sampling mechanism which allows for the verification of the expected behaviour of individual agents within a group.

Efficiency: Simulation runs can be time-consuming and expensive. In order to avoid unnecessary computation, it is thus important for a runtime verification approach to report results as early as possible. The 'runtime' nature of the model checking algorithms together with the idea of interleaving simulation and verification ensures that our framework reports verification results and stops the simulation as soon as a definite answer to a given property has been found. Verification time so becomes a direct function of the satisfiability or refutability of a property (with respect to the number of states necessary) which allows for the verification of certain properties with a high level of accuracy in a short amount of time — even for large-scale systems.

1.5 Publications and commercial use

Work described in this thesis has been published as the following peer-reviewed publications:

- Preliminary work such as (i) typical characteristics of agent-based simulations (see Chapter 3), (ii) requirements for a formal analysis process which have motivated the work described in this thesis, and (iii) a formal approach to reachability analysis of agent-based simulations which has, however, not been pursued further, has been presented at the '*1st International Workshop on Verification and Validation of Multi-Agent Models for Complex Systems (V2CS)*' and published in [118, 119].

- The idea of applying approximate model checking to large-scale agent-based simulations, an extension of LTL, referred to as *probabilistic LTL with arithmetic expressions (PLTLa)*, which forms the basis of simLTL (see Chapter 5), an offline model checking algorithm for PLTLa which forms the basis of the offline algorithm for simLTL (see Chapter 6) as well as a preliminary version of MC^{2}MABS (see Chapter 8) have been presented at the '*14th International Workshop on Multi-Agent-Based Simulation (MABS)'* and published in [120, 121].

The first version of MC^{2}MABS described in [121] also forms a central part of the validation process at Sandtable Ltd[4], a London-based company specialising in the development of large-scale commercial agent-based simulations, and has been used successfully in real-world projects.

[4] http://www.sandtable.com

Chapter 2

Background

This chapter introduces the theoretical concepts which are relevant for this work. Starting with a description of agents and multiagent systems in Section 2.1, we continue with an overview of computer simulation with a particular focus on agent-based modelling in Section 2.2. Chapter 3 makes heavy use of formal specification techniques, particularly the Z notation and Communicating Sequential Processes (CSP), both of which are introduced in Section 2.3. Section 2.4 introduces verification techniques with a particular focus on different variants of model checking and runtime verification. And finally, Section 2.5 considers different verification approaches for both probabilistic and non-probabilistic multiagent systems, again with a particular focus on model checking.

2.1 Agents and multiagent systems

This section gives a brief introduction into the notion of agency in Section 2.1.1 and the definition of multiagent systems in Section 2.1.2.

2.1.1 Intelligent agents

Despite huge efforts and impressive ongoing success in agent research, there is still no common agreement about some of the fundamental, underlying concepts — most notably the question what an ‘agent’ is and what constitutes its ‘intelligence’. This disagreement resembles the situation in general artificial

intelligence (AI); the AI community has been dealing with the development of intelligent systems for more than 50 years now — with impressive results — yet there is still no satisfying agreement upon the notion of intelligence, the most fundamental concept underlying almost all AI-related research efforts. As Wooldridge and Jennings point out, the lack of agreement upon the notion of an agent is not necessarily a problem "if many people are successfully developing interesting and useful applications" [245]. For a number of reasons, however, it is undoubtedly desirable to have some common understanding of the characteristics of agency and, for example, the distinction between agents and objects [98] or other types of software programs [95]. One potential risk of a lack of agreement is that the agent concept might eventually become a 'noise' term, resulting in confusion within the research community [245].

Despite there not being any official definition, several attempts to define agency have been made in the past. Wooldridge and Jennings propose the following definition:

> "An agent is a computer system that is situated in some environment, and that is capable of autonomous action in this environment in order to meet its delegated objectives."

In this definition, two aspects are mentioned which are widely accepted to be central to the notion of agency: *situatedness* and *autonomy*. Wooldridge and Jennings further refine their definition by differentiating between a *weak* and a *strong* notion of agency, a distinction which can also be found in general Artificial Intelligence [212]. For an agent to fulfil the requirements of weak agency, they propose a list of four characteristics: *autonomy*, *social ability*, *reactivity* and *proactiveness*. Strong agency further adds mental components like *beliefs*, *desires*, *intentions*, *knowledge* etc. to the list. Since Wooldridge and Jennings' definition is widely recognised in the research community, the four characteristics are briefly explained in the following paragraphs.

Autonomy defines an agent's ability to act on its own, without human intervention and to have control over its internal state and action. *Social ability* enables an agent to communicate with other entities (e.g. other agents or humans) using some kind of language; a major branch of agent research is therefore concerned with the problem of developing knowledge representations as well as communication languages in order to enable agents to exchange information and understand each other. *Reactivity* refers to an agent's ability to act within an environment; an agent should be able to perceive its environment through sensors and return information to the environment using its effectors. *Proactiveness* refers to an agent's ability to take initiative instead of just reacting in a purely reflex-based manner. Mental components like *goals* and *desires* play a fundamental role in this case since they help an agent to move from where it currently is to its desired state by choosing appropriate actions.

It is important to mention, however, that the notion of agency in the context of agent-based *modelling* and *simulation* (described in more detail in Section 2.2.2 below) is significantly weaker than it is in the more general area of multiagent systems. Simulations as tools for scientific discovery exhibit significantly different characteristics than systems constructed for the purpose of practical problem solving. Whereas, in the latter case, agents are designed as intelligent, autonomous and often adaptive problem solvers (as indicated above), their simulation-specific counterparts can be seen as mere 'sketches', i.e. abstract representations, of 'real' agents. In fact, simulation agents bear, in most cases, far more resemblance to simple software objects than to intelligent entities. As opposed to advanced concepts such as real autonomy and sophisticated reasoning capabilities, for example, efficiency, lightweightedness and analytical tractability are typically far more important in the simulation space. As a consequence, simulation-based agents are often implemented as simple key-value stores and the 'intelligent behaviour' reduces to mere rule- or threshold-based state changes. Nevertheless, conceptually, the idea of *autonomy* as an agent's capability to make decisions on its own, the entities' *situatedness* within an environment which they are able to perceive and to react upon and the ability to *adapt* to changing circumstances in the environment are still present in most simulations and thus justify the label of agency, albeit in a rather more abstract way than it is traditionally the case.

2.1.2 Multiagent systems

When considering systems of distributed entities, it is often useful to move to a higher level of abstraction and think of their properties in *anthropomorphic* terms [238]; it may, for example, be more suitable to view processes which are exchanging information as entities *talking* to each other; instead of the *information* stored by a process, it might be more useful to consider its *knowledge*. As described above, these are precisely the concepts related to the idea of agency. As soon as systems of distributed entities are viewed as sets of (possibly intelligent) interacting agents rather than as purely technical processes, one thus starts to deal with a *multiagent system*.

In general, a multiagent system can be understood as a set of distributed and interacting agents which are situated in an *environment* [243]. Multiagent systems share many characteristics with distributed systems; in fact, each distributed system can, in theory, be seen as a multiagent system. The opposite, however, is not true. As opposed to general concurrent and distributed systems, there is often no global architecture imposed upon the set of agents. This may, for example, be due to the fact that the individual agents belong to different companies and do not want to disclose all of their internals. Furthermore,

multiagent systems are often *open*, i.e. they allow agents to join and leave the system as they wish. And, due to the heterogeneous nature of the system and the autonomy of their constituents, there is typically *no global goal* in a multiagent system; agents are autonomous and pursue their own agenda which is determined by their knowledge, their personal goals and their available plans.

Due to the social nature of a multiagent system, a number of issues emerge [79]. First, agents are *situated*; in order to make reasonable decisions about the world they inhabit, they need a *model* of it, i.e. a representation of their knowledge about the environment, possibly including the knowledge about other agents, too. Second, agents may need to *share plans* in order to achieve individual or common goals. Third, agents have *social relationships* with each other which leads to questions about concepts like *norms*, *obligations*, and *prohibitions*. Fourth, agents need to *interact* with each other; in order to decide about how to perform interactions and how to assess their potential outcome, agents need models, i.e. formal representations, of each other. And finally, agents also need to *communicate* in order to exchange information or to coordinate their actions.

Research in the area of multiagent systems is highly interdisciplinary in nature and includes topics as diverse as agent architectures, distributed problem solving, multiagent learning, negotiation, argumentation and agent-oriented software engineering, to name but a few [243]. In the area of agent-based simulation — which is the focus of this work — computational agents are used to model real-world phenomena, e.g. human populations or socio-technical systems, with the overall goal of understanding better their dynamics, explaining certain phenomena, and making predictions about their future behaviour. Agent-based simulation is described in Section 2.2.2.

2.2 Computer simulation

One of the great benefits of cheap modern computer hardware is the possibility to use simulation as a virtual testbed for the exploration of various different scenarios. Shannon describes simulation as "the process of designing a model of a real system and conducting experiments with this model for the purpose either of understanding the behavior of the system or of evaluating various strategies (within the limits imposed by a criterion or set of criteria) for the operation of the system" [221]. Computer simulation makes it possible to explore complex real-world scenarios, determine correlations or conduct what-if analyses in an efficient and cost-effective way. This is particularly interesting when real-world experiments cannot be carried out, e.g. because of financial, legal or ethical constraints.

2.2.1 Types of simulation

In the literature, various different simulation classifications can be found [111, 203]. A useful first starting point is to categorise existing approaches using the following taxonomy:

- Numerical simulation
- Discrete-event simulation
- Monte Carlo simulation
- Agent-based simulation

Numerical simulation approaches deal with the simulation of systems — often physical processes [49] — described as sets of differential equations (in the continuous case) or difference equations (in the discrete case). Numerical simulation of differential equations is necessary if analytical solutions cannot be obtained. A typical area of application is *Computational Fluid Dynamics (CFD)* [93] where dynamics of fluid flows like wind, oil, or water are modeled and simulated [230]. *Difference equations* represent discrete parallels to differential equations where timesteps are implicitly embedded. They are useful for modelling discrete systems as well as approximations of continuous systems. A related field is that of *system dynamics* which focusses on the modelling of *feedback loops* in complex systems in order to understand the emergence of nonlinear behaviour. System dynamics models can be formulated as systems of differential equations and solved using numerical simulation.

Discrete-event simulation describes the temporal dynamics of a system based on events which occur at discrete time steps and influence the state of the system. According to Banks, a number of typical components can be identified within a discrete-event simulation [18]. The *model* represents the system under consideration in a formal, abstract way, for example as a finite-state automaton. In many cases it is also necessary to represent parts of the real-world system explicitly as entities within the system. Depending on their complexity, they might themselves contain attributes which change during the simulation and define the entity's *state*. The union of the single entity states represents the *global* or *system state*. In contrast to *activities*, which represent time periods with a definite length, *delays* can be indefinitely long. Examples of activities are the service time of a machine (which is probably modelled as an entity within the simulation) or the time a human worker needs to finish a given task. The time a worker needs to arrive at a broken machine represents a delay since it might be influenced by several other factors (e.g. other broken machines) and is thus not known in advance. The flow of time within the

simulation is represented by a clock variable which is incremented in discrete time steps. Based on their temporal order, events are removed from the list of future events at the time step at which they occur and used to trigger actions. In contrast to fixed-time events, some events may be probabilistic which means that their occurrence in time is random across the course of the simulation. For this purpose, a pseudo random number generator generates a sequence of numbers based on a given probability distribution (e.g. an exponential or a Poisson distribution) which is then used to specify the time of occurrence of the probabilistic event.

Instead of simulating a system based on a discrete temporal subdivision, *Monte Carlo simulation* uses a sufficiently high number of random samples in order to calculate the solution to a problem [205]. The Monte Carlo method is generally an efficient and powerful method to solve complex problems which cannot (or only less efficiently) be solved using conventional analytical techniques. Examples of problems from different areas which can be solved using Monte Carlo simulation are portfolio analysis [77], the solution of complex differentiation and integration problems [103], the estimation of the reliability of mechanical components [46] and reachability analysis for formal models in software engineering [22].

Agent-based simulation [169] uses populations of autonomous interacting agents to model various phenomena, typically the dynamics of complex systems. Since the focus of this work is on agent-based simulation, a more detailed description is given in Section 2.2.2.

It is important to note that the categories mentioned above are not mutually exclusive. It is, for example, perfectly possible for an agent-based model to incorporate aspects of discrete-event simulation [247] or differential equations [17].

Apart from the classification given above, another common distinction between different types of simulations is whether they are *deterministic* or *stochastic*. Generally, a simulation can be seen as a function (albeit in most cases a very complex one) which maps input values to output. In deterministic simulations, input values are fixed and the simulation will produce the same output values across multiple runs. Kelton refers to this scenario as *DIDO (deterministic in, deterministic out)* [133]. In the stochastic case, input values are not fixed but instead they follow a probability distribution. For example, in the case of a queuing simulation, the time required to process a customer typically follows a normal distribution. Since the input values are probabilistic, the output will also be probabilistic and multiple simulation runs will therefore produce different results. Kelton calls this scenario *RIRO (random in, random out)*. Randomness in the simulation output is an important factor which needs to be taken into account when

analysing the results; in order to cope with statistical variance, multiple simulation runs need to be performed and statistical analysis needs to be applied in order to gain confidence in the results.

The difficulty of simulation output analysis also depends on another distinction which can be drawn between different types of simulations, namely whether they are *terminating* (finite) or *steady-state* (infinite). Terminating simulations are designed to run for a specific period of time, e.g. to simulate a production day in an engineering plant. After the simulation has finished, all results are available and can be analysed. In the case of stochastic simulations, multiple samples can be created and executed in order to cope with the randomness in the output. In contrast to terminating simulations, steady-state simulations may run forever. Since there is only a single execution which produces results continuously, it is not possible to use output from different independent samples for the purpose of statistical analysis. Coping with randomness in steady-state simulations is a difficult problem which has been addressed extensively in the literature [214]. One proposed solution is to run the simulation multiple times for a (sufficiently) long period of time instead of running it infinitely. Alternatively, the underlying time series can be checked for ergodicity and stationarity using statistical tests such as Dickey-Fuller [78].

Since randomness is inherent to most simulations, a substantial amount of research deals with related problems, e.g. experimental design [134, 135], distribution of input values [152] and efficient output analysis and validation [215].

2.2.2 Agent-based simulation

An important area of application of multiagent systems is the simulation of complex adaptive systems. This section gives a brief introduction into agent-based modelling with a particular focus on social simulation. The description has been deliberately kept general; a more detailed description of the characteristics and constituents of agent-based models (both informal and formal) is given in Chapter 3.

2.2.2.1 General characteristics

Due to an agent's ability to communicate and exchange information, its autonomy in terms of decision making and its adaptability to the environment, multiagent systems are capable of exhibiting complex overall behaviour. Regarding the simulation and modelling of real-world phenomena, this capability can be particularly attractive. Many real-world systems, e.g. human societies, the financial market or the weather, are highly complex in nature and therefore difficult to understand, explain and predict.

However, understanding their dynamics and being able to evaluate different scenarios by conducting *what-if* analyses is a critical requirement in many areas. When dealing with complex systems, classical analytical approaches like equation-based models (e.g. system dynamics) are often not sufficient [196]. As opposed to agent-based simulation which starts with the local rules of the individuals, equation-based modelling (EBM) can be seen as a 'top-down' approach. EBM puts a particular focus on the global level, relates observables (i.e. measurable characteristics) with each other and describes their evolution over time. Since it does not represent individuals and their relationships explicitly within the model, EBM is not capable of replicating feedback mechanisms and non-linear effects that result from interactions between the components. Agent-based simulation, on the other hand, is well-suited for the simulation of complex systems due to its individual-based nature and its support for heterogeneity. According to Parunak *et al.*, "agent-based modelling is most appropriate for domains characterised by a high degree of localisation and distribution and dominated by discrete decision. Equation-based modelling is most naturally applied to systems that can be modelled centrally, and in which the dynamics are dominated by physical laws rather than information processing" [196]. The individual-based approach is particularly attractive for simulation purposes since it allows the simulator to study the correlation between individual interaction and global behaviour and provide insights which cannot be determined otherwise. This makes agent-based models a powerful alternative to classical approaches for the simulation of complex systems.

Even though agent-based simulation is quite similar to classical discrete-event simulation and shares some of its characteristics (e.g. both comprise multiple individual entities and focus on the temporal dynamics of the system, and the system may, in both cases, progress on the basis of events rather than discrete time steps), there are also differences [50]. One important difference is that entities in a discrete-event simulation do not exhibit autonomous behaviour and hence do not make decisions on their own; instead, they respond to given input on a tick-per-tick basis in a purely reactive way. As indicated in Section 2.1.1, however, this is common to many simulations and it is debatable where the boundary between pure reactivity and autonomous decision making is to be drawn. Furthermore (and more importantly), entities in a discrete-event simulation do not interact with each other. Without this capability, it is difficult or even impossible to reproduce many emergent phenomena typical for complex systems that can be observed in reality. Due to these differences and similarities, Chan *et al.* refer to agent-based simulation as "a hybrid discrete-continuous simulation model with proactive, autonomous, and intelligent entities" [50].

Due to its general applicability to various different problem domains, agent-based simulation has been

- Business and Organisations
  - Manufacturing
  - Consumer markets
  - Supply chains
  - Insurance
- Economics
  - Artificial financial markets
  - Trade networks
- Infrastructure
  - Electric power markets
  - Hydrogen economy
  - Transportation
- Crowds
  - Human movement
  - Evacuation modelling
- Society and culture
  - Ancient civilisations
  - Civil obedience
- Terrorism
  - Social determinants
  - Organisational networks
- Military
  - Command & control
  - Force-on-force
- Biology
  - Ecology
  - Animal group behaviour
  - Cell behaviour
  - Sub cellular molecular behaviour

TABLE 2.1: Application areas for agent-based modelling (adapted from [168])

employed successfully in a variety of different areas such as economics, social science, biology and consumer marketing. A comprehensive overview can be found in Macal and North's tutorial on agent-based modelling and simulation [168] (see Table 2.1).

A useful way of looking at agent-based simulation models is to classify them according to purpose and their reliance upon external data as, for example, done by Heath *et al.* [115]:

Generators: Theoretical models with the overall goal of confirming behavioural hypotheses and assumptions and identifying mechanisms

Mediators: Models that incorporate both behavioural assumptions and real-world data with the overall goal of identifying mechanisms to a certain extent and performing short-term predictions

Predictors: Purely data-driven models with the overall goal of performing medium- to long-term predictions

Social science research typically deals with generator models; classic examples are Schelling's model of racial segregation [218] and Epstein's *Sugarscape* model [89]. Simulations of technical systems such as robot swarms as well as models of consumer marketplaces which are highly driven by (historical) data, on the other hand, can be attributed to the third category. It is important to note that the categories are not

mutually exclusive and should rather be seen as a continuous spectrum. Generator models typically have a strong *explanatory* flavour, whereas predictors — as the name suggests — mostly focus on *prediction* and *forecasting*.

Due to its individual-based and bottom-up approach, the agent-based paradigm is particularly powerful and appealing for the construction of explanatory models. This has also been shown empirically by Heath *et al.* who surveyed a wide range of published works in the agent-based modelling literature and found that, in the analysed time period (1998-2008), all models could be either classified as generators or as mediators [115]. Nevertheless, especially in the context of complex and potentially safety-critical socio-technical systems, agent-based modelling is also increasingly showing promising results for simulating, predicting and analysing their runtime behaviour [38, 39, 179, 236].

2.2.2.2 Agent-based social simulation

The study of human behaviour on both the individual (micro) and societal (macro) level is traditionally the domain of the social sciences. Psychologists, sociologists, anthropologists, economists and political scientists all share an interest in understanding the mechanisms underlying human behaviour in general and the processes that govern human decision making in particular. Despite ongoing efforts and success in understanding brain activities and behaviour on an individual basis, the emergence of behavioural patterns and higher-level phenomena such as the adoption of new technology or the spreading of opinions remain challenging to explain and predict. The main reason for the difficulties lies in the complex nature of the underlying system. Unlike many physical systems, societies (and particularly human societies) are highly complex and adaptive; they consist of interconnected individuals, each of them equipped with its own local behaviour and the ability to communicate with other individuals; information exchange can influence an agent's state or trigger behavioural actions which can themselves trigger further actions or state changes. These dynamics may introduce complex *feedback loops* which have a significant influence on the complexity and nonlinearity of the resulting behaviour, particularly the emergence of global patterns that are typical for complex (adaptive) systems like human societies [187]. As a consequence of the emergent character of the system, it is extremely difficult and often impossible to understand the global behaviour by following the classic reductionist approach and by simply segmenting the population into their constituents in order to analyse their local behaviour. Emergence, a principle which has probably best been summarised by Aristotle's famous citation "the whole is more than the sum of its parts", is a central phenomenon of complex systems and has been studied extensively in the literature [108, 104, 19, 25]; especially in the area of social systems, emergence has received a great deal

of attention. As Gilbert points out, "markets emerge from the individual actions of traders; religious institutions emerge from the actions of their adherents; and business organisations emerge from the activities of their employees [...]" [105]. To add a further level of complexity to a society's dynamics, human individuals are able to *perceive* emergent patterns and act accordingly. Complex global patterns emerging from individual behaviour (e.g. norms or conventions) are thus themselves able to influence the individual's behaviour which introduces another, even higher-level, feedback loop. To complicate things further, social networks are highly dynamic since individuals may leave, new individuals may join and connections are thus subject to change over time.

Understanding and influencing social behaviour is an important requirement for various different areas, ranging from academic research to more commercially-driven activities in the private sector such as marketing research projects. Due to the complexity and nonlinearity mentioned above, many complex social processes cannot be represented as equations and traditional mathematical approaches are therefore often insufficient to analyse societies [158]. In contrast to equation-based methods, social simulation takes a computational approach to model and simulate social interactions in a bottom-up way. Since its beginning in the 1970s, social simulation has grown significantly and attracted much interest in the wider research community. In an influential paper, the political scientist Axelrod describes social simulation as a "third way of doing science" which complements deductive and inductive approaches [9]. He emphasises simulation as a powerful tool for prediction, proof and discovery in the scientific world as well as as for problem solving, training, entertainment and education. The particular appeal of simulation for research lies in its usefulness for discovering correlations by creating models based on simplified assumptions and analysing their behaviour. According to Axelrod, "the simpler the model, the easier it may be to discover and understand the subtle effects of its hypothesised mechanisms". In contrast to real experiments, where numerous subtle factors might influence and distort the results, simple computational models provide the advantage of containing only what has been implemented by the developer.

With the availability of affordable computer hardware (particularly the advent of personal computers in the 1980s) and the constantly increasing computational power since then, creating artificial individuals and societies and simulating their behaviour has provided an interesting alternative to classical social science. Complexity issues as well as the necessity to comply with ethical and legal regulations place strict constraints on real-world experiments. Social simulation enables researchers to bypass these restrictions and to carry out experiments and *what-if* analyses that would be impossible, unethical or even illegal to perform in reality.

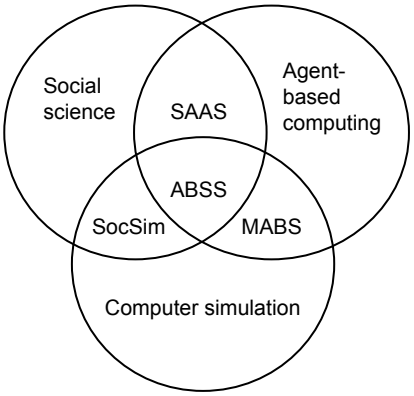


FIGURE 2.1: The intersections of the three areas defining agent-based social simulation [68]

Social simulation is a large research area which can be subdivided into several subsections. One of these subsections is *agent-based social simulation* which follows the principle of interacting, autonomous components and inherits the characteristics from agent-based simulation described above. A good classification of agent-based social simulation into its surrounding research areas has been given by Davidsson [68] (see Figure 2.1). He places agent-based social simulation at the intersection of three major research areas: *social science*, *computer simulation* and *agent-based computing*. Davidsson also identifies areas which are situated at the intersections of two of the three areas mentioned above yet without being a full part of agent-based social simulation: *social simulation* (SocSim), *social aspects of agent systems* (SAAS) and *multiagent-based simulation* (MABS). Social simulation deals with the simulation of social societies in general and typically uses simple techniques such as cellular automata [187]. According to Gilbert and Troitzsch, the development of social simulation over the past fifty years can be roughly divided up into three phases: *dynamical systems*, *microsimulation* and *adaptive agent models* [110]. During the first phase, models typically consisted of sets of differential equations and tried to describe changes over time. During the second stage, the focus began to shift towards the individual level, but still without employing adaptivity or explicitly modelling interactions between the entities. As Macy and Willer point out, "these models remain equation-based, like the earlier dynamical systems models" [170]. The third phase began with the advent of personal computers in the 1980s and refers to agent-based models for social simulation. Social aspects of agent systems (SAAS) lies at the intersection between social science and agent-based computing. It deals with social aspects like norms, cooperation, competition, coordination and organisations and is today one of the central research areas within the wider field of multiagent systems, as indicated in Section 2.1.2. Finally, multi-agent-based simulation (MABS) — or simply agent-based simulation, as we refer to it in this work — uses agent

technology for simulation purposes; it has been described in Section 2.2.2.

Agent-based social simulation provides a powerful tool to simulate individuals and societies over a long period of time. This offers researchers the opportunity to gain profound insight into long-term developments, e.g. in contrast to game theory which puts a special focus on the simulation of strategic situations and typically employs a very small number of participants only. As Midgley points out, "even where game theory has been employed to model the interactions among different types of agent, this has often been for 'one-period' or 'two-period' games rather than the interactions over multiple periods that characterise the setting." [186]. As opposed to two-period or multi-period games, in a one period game only a single action-response cycle is considered. A famous example of a one-period game is the Prisoner's Dilemma [7]. *Empirical game theory* as a combination of game-theoretic analysis and simulation represents an interesting approach for cases where classical analysis is intractable [200]. It can be seen as a special type of agent-based simulation in which interaction happens in discrete time steps within randomly chosen pairs of agents.

Similar to other software systems, correctness also plays an important role in the development of (agent-based) simulations. The next section discusses the notion of correctness against the background of simulation and highlights some of the conceptual difficulties and ambiguities in this context.

2.2.3 Correctness in a simulation context

This work focusses on the problem of correctness checking in the context of agent-based simulation. However, in the context of modelling and simulation, the notion of correctness can be surprisingly hard to define. In order to deem a model correct, one needs to take into account its *purpose*, i.e. the goal a modeller strives for when constructing a model. When considering modelling in general, prediction is often stated as the predominant goal. As argued by various researchers, however, prediction is just *one* of a wide range of reasons for constructing models. For example, in his seminal paper "Advancing the art of simulation in the social sciences", Axelrod lists seven reasons for constructing models in the social sciences [9]; in "Why model?", Epstein lists 17 purposes which range from *explanation* to *guiding data collection* to *training* and *education* [87]; and in "What are models for?", McBurney lists 12 reasons for modelling [178]. An overview is shown in Table 2.2.

Correctness is an essential requirement for any software system and agent-based models are no exception. The literature on quality assurance in the area of software in general traditionally distinguishes

Epstein [87]	Axelrod [9]	McBurney [178]
- Predict	- Prediction	- Understand natural reality
- Explain	- Performance	- Predict natural reality
- Guide data collection	- Training	- Control natural reality
- Illuminate core dynamics	- Entertainment	- Understand an existing human model or artefact
- Suggest dynamical analogies	- Education	- Predict an existing human model or artefact
- Discover new questions	- Proof	- Understand, predict or control a future human model or artefact
- Promote a scientific habit of mind	- Discovery	- Provide a locus for discussion
- Bound (bracket) outcomes to plausible ranges		- Resolve trade-offs
- Illuminate core uncertainties		- Structure thinking
- Offer crisis options in near real-time		- Train people
- Illuminate core dynamics		- Assess the modellers
- Demonstrate tradeoffs / suggest efficiencies		- Play
- Challenge the robustness of prevailing theory through perturbations		
- Expose prevailing wisdom as incompatible with available data		
- Train practitioners		
- Discipline the policy dialogue		
- Educate the general public		
- Reveal the apparently simple (complex) to be complex (simple)		

TABLE 2.2: Functions of models according to Epstein [87], Axelrod [9] and McBurney [178]

between *verification* and *validation*. Whereas the former is targeted towards a system's correctness with respect to its specification (i.e. its correct implementation), the latter ensures a sufficient level of accuracy with respect to the intended application domain. In the context of modelling and simulation, the distinction between verification and validation becomes more complicated; it is thus often more useful to talk about *internal* and *external validation* instead [199, 29]. Papers about agent-based modelling often also mention *calibration* as a third important step in the quality assurance process [48]. Given the heterogeneity of those models (all agents might have different attributes and behaviours) and the often large amount of agent-independent system-level parameters which further influence the behaviour of the system, careful calibration of their initial values is, in fact, critical for correctly interpreting the simulation outcome. A good overview of calibration, verification and validation against the background of (geospatial) agent-based modelling has, for example, been given by Castle and Crooks [48]; a discussion of the terminological issues surrounding the different notions of veracity and validity has been given by David [67]. We revisit the three notions (verification, validation, and calibration) below in order to clarify the terminology being used in this work.

2.2.3.1 Verification

As mentioned above, verification is also often referred to as *internal validation*. In a general software engineering context, verification is typically understood as checking the correctness of the implemented *code*. On a basic level, verifying an agent-based model is no different from verifying any other piece of software and well-established software engineering techniques such as static analysis, testing or informal code reviews can be used [15, 48]. However, what distinguishes the verification of agent-based simulations from that of conventional software systems is their focus on *emergent phenomena* which are, by definition, not (or only in certain cases) reducible to the behaviour of the constituents of the system. Furthermore, causal mechanisms may exist between different levels, e.g. from the emergent phenomenon to the individual constituents (*downward causation* [44]). As a consequence, apart from the verification of individual components, there is a strong need for the verification of the model *as a whole*. We refer to this more holistic view on verification as *model verification* [14]. Rather than with mere freedom from errors in the underlying code, model verification is concerned with the correctness of the model's *internal dynamics* or against an abstract specification, e.g. a conceptual model. As opposed to purely statistical models, agent-based models put a strong focus on *theory*; *causal mechanisms* that are based on empirical evidence and are incorporated into the model as behavioural rules play an important role for the dynamics of the model. Model verification can thus be characterised as the assessment of the *model-theory link*, i.e. the model's *internal properties*, without taking into account the relationship between the 'model world' and the 'real world' [199]. McKelvey describes verification as a process to ascertain the *analytical adequacy* of the model, i.e. the fact that "the model [...] correctly produces effects predicted by the theory" [180]. According to Bharathy and Silverman, "the verification process attempts to re-create the very empirical evidence used for constructing the model" [29]. If a model has been shown to be internally valid, a modeller can thus have confidence in the correct functioning of its internal causal mechanisms.

It is important to note that, due to the emergent nature of many phenomena of interest in agent-based modelling, code verification — albeit necessary — is generally not sufficient for showing the internal validity of a simulation model. Interactions among agents, between groups of agents and the environment, randomness and the emergence of higher-level phenomena complicate the analysis process and may often render straightforward application of conventional software engineering techniques such as testing difficult or even impossible. Errors in the underlying code *can* but do not necessarily *have to* cause errors in the temporal dynamics of the model at runtime. Likewise, errors in the temporal dynamics of the model need not necessarily be caused by errors in the code; they could equally well result from

flaws in the conceptual design of the model (the code being perfectly correct in this case). Consider, for example, out-of-bounds access to an array in a programming language like C or C++. Depending on the value that is being read, the invalid access may not cause any deviations in the runtime behaviour at all. Nevertheless, out-of-bounds access is clearly a critical error that can (and should) be eliminated prior to the productive use of the model. On the other hand, consider a simulation model of a robot swarm. In order for the swarm to be as efficient as possible (e.g. with respect to energy consumption), appropriate *coordination mechanisms* need to be integrated [160]. In such a scenario, robots may, for example, be expected to reduce their own activity whenever they think the environment is overcrowded (i.e. whenever they perceive a large number of other robots in their direct neighbourhood) in order to avoid interferences. What complicates the verification process here is the downward causation between the swarm as a whole and the individual robots. This particular causal relationship can be seen as a central mechanism that the 'theory' behind the swarm model relies upon that should thus be checked as part of the verification process. Due to the emergent nature of the cause (overcrowding), however, the correctness check cannot be performed on the code level alone. Despite all individual components being potentially error-free and working as expected, the overall causal mechanism may still fail — often simply because the interrelationships between mechanisms operating on different levels are too complex to be fully understood by a human modeller. In order to detect problems related to higher-order causal relationships, it is crucial to assess the correctness on the *model level*.

An important technique for verification is *sensitivity analysis*, which investigates the effect of changes to model parameters on the overall dynamics of the model. In this way, parameters which should not but do have an effect, as well as parameters which should have but do not have an effect on a certain behaviour can be found. In theory, each parameter (and ideally even parameter combinations) should be tested but, depending on the parameter space, this might not be feasible. A particularly effective method for finding implementation errors is *replication*, i.e. reimplementation of the existing system, ideally using a different programming language. In the simulation space, replication and alignment of different models is referred to as *docking* [10]. It is obvious, however, that full reimplementation is only a realistic option if the model is either sufficiently simple or the development is not subject to typical resource constraints.

The usefulness of internal validation is illustrated in more detail in Chapter 9 which contains a comprehensive case study in the area of swarm robotics. Starting with a very simple macroscopic and probabilistic representation of a robot swarm, it is gradually expanded into a fully-fledged rule-driven, agent-based model. In each of the steps, several internal validation experiments are performed in order

to assess the correctness of the incorporated mechanisms. For example, an obvious assumption of the 'theory' behind the implemented model is that there is a positive causal relationship between the field of vision of an individual agent and the amount of food that the agent is able to detect and collect. Another expected causal mechanism is that increased competition between agents has a negative effect on the overall efficiency of the swarm. Due to the emergent nature of the effect in the first example and the cause in the second one, it is impossible to assess the correctness of the causal mechanisms purely on the code level. Although the detection of *true* causal mechanisms is very hard (for a discussion, see Section 7.6), answering different types of correctness properties can help to shed light on the underlying dynamics of the model and *indicate* potential causal relationships. As described in Section 7.6, a probabilistic view on causality states that, broadly speaking, B is causally dependent on A if the presence of A *raises the probability* of B. As a consequence, being able to estimate the probability of events is an important requirement for causal analysis. For example, the analysis of *transition probabilities* can help to shed light on causal relationships between states that an agent or a group of agents can be in; by varying external parameters of the model, i.e. independent variables, and assessing the effect on the probability of dependent variables, insights into causal relationships can be obtained. This is described in more detail in Section 7.5.2 and illustrated in various places in the case study in Chapter 9.

2.2.3.2 Validation

As opposed to verification, *validation* in the context of modelling is concerned with the *model-phenomenon link*, i.e. with how well its output corresponds with the real-world phenomenon of interest. In contrast to the assessment of internal mechanisms described above, this process is often referred to explicitly as *external validation* [199, 29]. McKelvey refers to external validity as *ontological adequacy*, i.e. "the ability of the model to represent real-world phenomena defined as within the scope of the theory" [180].

Whilst calibration (described below) aims to find appropriate initial values such that the error between the model output and given training data is minimised, an important goal of validation is to ascertain that the model has not been overfitted to the training data. Since the focus of this work is on the internal rather than on the external validation of simulations here, we shall not further elaborate upon the exact notion of ontological adequacy and its confirmation at this point; for a comprehensive overview, please see the relevant literature [216, 177, 241]. At this point, it is sufficient to assume that external validation requires the existence of *empirical data*, for example historical sales records, which the outcome of the simulation is compared with in order to determine the level of correspondence. Validation is typically an empirical process: in order to deal with the enormous state space which is caused by the high number

of different behaviours that the model can exhibit and the variance in the output due to randomness in the model, comprehensive experiments need to be conducted. It is thus not surprising that a significant amount of the existing work on simulation validation is concerned with statistical analysis [139, 177].

It is clear that different types of model may require different levels of internal and external validation, respectively. For example, due to their lack of underlying theory, very simple, purely probabilistic models (such as the one described in Section 9.3) may only require very basic internal validation; complex rule-based models where the behaviour of individual agents is based on several behavioural assumptions (as is, partly, the case for the model described in Section 9.5), on the other hand, may contain a variety of causal mechanisms whose individual correctness and correct interplay may need to be ascertained. Extensive *external* validation is particularly important in cases where the correspondence between the model and the phenomenon of interest cannot be established formally. An application area of agent-based modelling that is highly reliant upon validation is *social simulation*, i.e. the simulation of social phenomena in human populations using a computational approach, described in Section 2.2.2.2 above. Whenever simple agents are supposed to represent complex human beings, showing their correspondence is essential for establishing confidence in the model. In this case, validation constitutes a major (if not the most important and comprehensive) task in the modelling process. In a purely technical domain, on the other hand, validation may be less problematic. Consider, for example, the simulation of a robot swarm. In this case, the internal states and rules of the individual robots are all known to the modeller and can be translated one-to-one into their corresponding software counterparts in the simulation. In this case, the model-phenomenon link is typically much easier to show and rarely the subject of extensive debate.

It is important to note that it is not always clear where the boundary between internal and external validation should be drawn. The theory underlying a simulation model often results from empirical evidence and incorporates *stylised facts*, i.e. general expectations about the real-world phenomenon. Often, those expectations are common sense-turned *a-posteriori* insights that result from decades or even centuries of empirical or analytical research and thus also possess an 'external flavour'. A typical example is the assumption that, in an epidemiological context, an infection rate of 100% may be highly unrealistic and should thus never be attained. It is clear that, despite not referring to a particular data set, this assumption establishes a clear connection between the simulation and the real world phenomenon. Especially in cases where internal validity is based upon such common sense assumptions, the differentiation between internal and external validity may become difficult.

According with McKelvey's definition [180], the basic difference between internal validation (verification) and external validation can be summarised as follows:

Internal validation (verification) targets the *model-theory link* and aims to ascertain the *analytical adequacy* of the simulation model. A particular focus is put on the *analysis of causal mechanisms*.

External validation targets the *model-phenomenon link* and aims to ascertain the *ontological adequacy* of the simulation model. A particular focus is put on the capability of the model to *reproduce empirical observations*.

The focus of this work is on *internal validation*. Although the techniques described in the following chapters may also be usefully employed for the purpose of external validation (see, e.g., Section 4.4), the focus is on the internal dynamics of the simulation models.

2.2.3.3 Calibration

Proper validation is often preceded by a *calibration* step. The main purpose of calibration is to find appropriate initial values for independent variables (i.e. model parameters) such that the error between the simulation output and empirical data w.r.t. some metric of interest is minimised. Calibration can thus be seen as a training step in which the model is fitted to one particular data set. Further validation ensures that the model has not been overfitted and that it generalises well enough in the presence of previously unknown data. Due to the heterogeneous nature of agent-based models where agent attributes are typically set on an individual basis, the calibration process can result in a complex optimisation problem. Given the often limited amount of real-world data for a particular phenomenon, calibration and validation are often intertwined processes.

It is important to stress that the notion of correctness is also dependent upon the *observational level* that one is operating on. The individual nature of agent-based modelling makes it suitable for fine-grained representations and simulations of complex phenomena. However, the fine level of granularity offered by an individual model requires careful assessment of the behaviour on different levels. It is useful to distinguish between three observational levels: *micro* (the individual level), *meso* (the group or organisational level) and *macro* (the system or aggregate level). Despite the output of the simulation being correct on the macro level – e.g. when analysing total sales figures resulting from a consumer marketplace simulation — the individual behaviour may still be incorrect. Conversely, despite the

behaviour of individual agents being correct with respect to given reference data, the resulting aggregate behaviour may not correspond with reality. And finally, despite both micro and macro level behaviour being correct with respect to given reference data, there may still be discrepancies at the meso level, for example within the behaviour of a particular age group of agents. Behaviours on different levels are clearly dependent upon each other. Quality assurance needs to take this into account and, ideally, verify, validate and calibrate the simulation model on all observational levels.

The previous paragraphs indicate that a definition of correctness in the context of simulations is not straightforward. First, the notion of correctness is dependent upon the purpose of the model: a model is correct if it satisfies its purpose. The purpose of the model, however, is often difficult to determine; it depends itself on the types of questions one wants to answer. It is not uncommon in modelling, however, that those questions are not known in advance but instead emerge from the insights obtained from the modelling process itself. The purpose of the model also depends on the domain one is working in (e.g. social science versus information technology); it determines the necessity of certain steps (e.g. validation) as well as their target (real-world or simulated system). And finally, correctness can (and should) be defined with respect to different levels (micro, meso, macro) and focus on different aspects of the system (code versus model internals versus representational accuracy). The complexity of the problem is reflected in the abundance of scientific literature about simulation verification, validation and analysis and the plethora of conferences, workshops and journals which are solely dedicated to this topic.

2.3 Formal specification

An important part of the research project involves the use of formal specification techniques. Among the range of available formal specification languages, we focus on the family of Z-based notations [228], a decision which was motivated by the following facts:

- Z has been used extensively to model multiagent systems and related concepts [79, 206].
- Z specifications and their underlying transition system semantics bridge the gap between high-level and low-level formalisms.
- Z has a strong mathematical foundation and supports formal proof.

- Z is an established formalism, is well understood and has been applied successfully to numerous real-world scenarios in an industrial context.

Within the category of Z-based notations, we focus on Object-Z [226], an object-oriented extension of the Z notation. In the following sections, we give a brief overview of the syntax and the semantics of Object-Z in Section 2.3.1, those of CSP in Section 2.3.2 and the integration of Object-Z and CSP in Section 2.3.3.

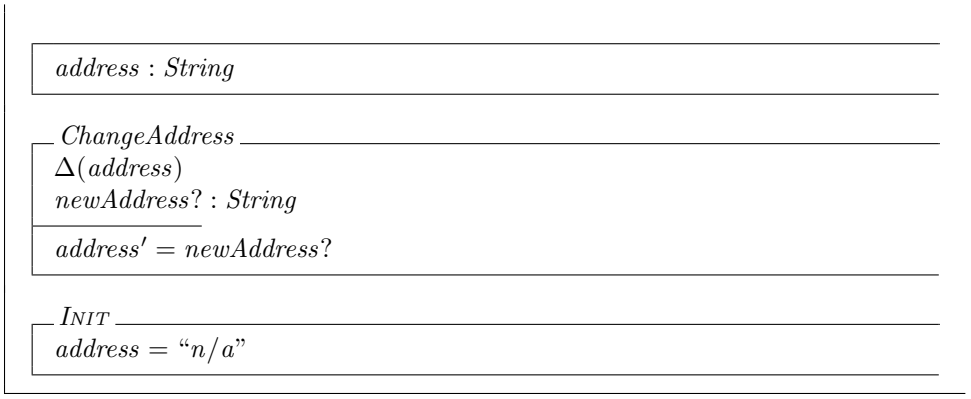
2.3.1 Z and Object-Z

The Z notation is a well-established formalism for the specification of computer systems; it was originally developed by Jean-Raymond Abrial in the 1970s and it first appeared in the book "Méthodes de programmation" in 1980 [185]; since then, it has been extended significantly, mostly at the Programming Research Group at the University of Oxford. A full description of Z is beyond this work, interested readers may consult the relevant literature [228, 242].

The basic component of the Z language is a *mathematical toolkit* which provides a notation for commonly recurring mathematical entities such as sets, relations, functions, and sequences. Z further offers a *schema language* which helps to structure specifications; a tabular overview of the mathematical toolkit and the schema language is given in Appendix A. Finally, Z also provides support for the process of *refinement* with the overall goal of translating a formal specification into executable code. Refinement is not relevant for this work and is therefore omitted. A good overview can be found in Woodcock's and Davies' book [242].

Object-Z [226] is an object-oriented extension to Z and augments its syntax with object-oriented concepts such as *classes*, *objects*, *inheritance* or *polymorphism*. For clarity, we only give a brief overview of Object-Z; for a more comprehensive description, please refer to Smith's original reference document [226]. A major difference between conventional Z and Object-Z is the introduction of a *class schema* which allows for the specification of encapsulated entities. A simple schema of a class representing a customer with an id, an address and a function to change the address is given below. It could, for example, be part of a customer relationship management system:

$$
\begin{array}{|l}
\hline
Customer[Person] \\
\lceil (ChangeAddress) \\
\mid \; id : \mathbb{N} \\
\end{array}
$$



Within a class schema, it is possible to specify the following components:

- inheritance relationships (in this case, $Customer$ inherits from class $Person$)
- constant (e.g. id) and non-constant member variables (e.g. $address$)
- member functions ($ChangeAddress$)
- initialisation of member variables ($INIT$) and
- the visibility of class members ($\upharpoonright$[1])

It is possible to make a distinction between constant and variable member attributes in Object-Z. The former are denoted by a vertical line (id in the example above), the latter are enclosed in a simple frame ($address$). Class operation schemas are similar to the operation schemas in conventional Z, with one difference: in Object-Z, operation schemas comprise a *delta list*, denoted by $\Delta(...)$, which contains those class attributes that are changed by the operation.

As described above, Object-Z also formalises advanced object-oriented aspects such as polymorphism. We do not use these concepts here and their description is thus omitted. Further information can be found in the full language description [226].

[1] All class members enclosed in the *visibility list* after $\upharpoonright$ are visible to the environment, i.e. *'public'*. We will, however, omit the visibility list in the class schemas in this document since we assume that all class members are visible by default.

2.3.1.1 Semantics of Object-Z

The semantics of Object-Z are based on two central models: the *structural model* and the *history model*. The main purpose of the structural model is to establish a correspondence between attributes and operations of Object-Z classes and states and transitions. Rather than its structure, the history model describes the *type* of an Object-Z class by describing all possible histories it can undergo. We do not describe the semantical models here, please refer to Smith's paper for further information [224]. We just mention two meta definitions below which relate the notion of Object-Z classes with states and events; the definitions are part of the structural model and described in Smith's paper.

The *state* of an Object-Z class is determined by the assignment of its variables at any time t. Variables consist of a *name* (or *identifier*) and a *value*. Let Id denote the set of all possible identifiers and $Value$ the set of all possible values. A state can then be defined as an assignment of values to ids:

$$State == Id \pfun Value \tag{2.1}$$

Transitions or *events* in the transition system correspond with operations in the Object-Z specification. An operation has a name and a (possibly empty) set of parameters. An event can then be defined as a tuple containing the operation name and an assignment of values to operation parameters:

$$Event == Id \times (Id \pfun Value) \tag{2.2}$$

The relation between Object-Z classes and states and transitions is an important concept for the integration of Object-Z and CSP described in Section 2.3.3.

2.3.2 Communicating Sequential Processes (CSP)

Communicating Sequential Processes (or CSP) is a formal language for the description of concurrent and interacting systems [125]. It belongs to the family of *process algebras*, other examples of which are Milner's *Calculus of Communicating Systems (CCS)* [189] or probabilistic variants such as Hillston's *Performance Evaluation Process Algebra (PEPA)* [122]. The syntactical fragment of CSP used in this work comprises only *processes*, *events* and a set of operators. A simple example of a CSP process specification is given below:

$$P = e \rightarrow P \tag{2.3}$$

Here, process P is able to engage in event e after which it will continue to act as P. The example shows the recursive nature of CSP processes which makes it possible to describe infinite (i.e. non-terminating) processes succinctly. '$\Box$' represents *external choice*. It offers the environment the possibility to engage in different events:

$$P = e \rightarrow P \,\Box\, f \rightarrow P \tag{2.4}$$

This simple process is able to engage in event e *or* in event f and will subsequently act as P (no matter whether it engaged in e or f). CSP also offers an *internal choice* operator '$\sqcap$' which represents the choice between different options that is made *by the process itself* in a *nondeterministic* way. Finally, processes and events can also have an *internal state*, i.e. they can carry data in the form of attributes. In order to express that, both the CSP process and the event can be *parametrised* as shown in the following example:

$$P(n) = e.n \rightarrow P(n+1) \tag{2.5}$$

Here, process P has an internal variable n. The process can be understood as engaging in event e which causes n to be incremented by 1. When parametrised in this way, events become *communication channels* over which data can be transferred. More precisely, in the example above, process P can engage in communicating value n on channel e; it can only synchronise with another process which is also able to engage in the communication over channel e. Synchronisation of P with the following process Q which is willing to engage in e but not in the communication of a value would thus not be possible:

$$Q = e \rightarrow Q \tag{2.6}$$

It is sometimes necessary to combine the communication of a value over a channel with external or internal choice as shown in the following example:

$$P = e.1 \rightarrow P \,\Box\, e.2 \rightarrow P \tag{2.7}$$

In this example, process P offers the environment the possibility to engage in the communication of value 1 *or* value 2 over channel e. This is a frequently recurring pattern which can be conveniently abbreviated as follows:

$$P = e?\{1, 2\} \rightarrow P \tag{2.8}$$

Since the choice of value is eventually made by the environment rather than by P itself, this can be considered an *input event*, i.e. process P is *receiving* a value over e. The opposite happens in case of an internal choice:

$$P = e.1 \rightarrow P \sqcap e.2 \rightarrow P \tag{2.9}$$

Here, the environment does not know what value communication on e process P is going to be able to engage in since the choice is made nondeterministically by P itself. This can be considered an *output event*, i.e. process P is *sending* a value over e; this is abbreviated as follows:

$$P = e!\{1, 2\} \rightarrow P \tag{2.10}$$

The major strength of process algebras lies in their possibility to describe the interacting behaviour of multiple processes in a compositional way. To this end, CSP provides two parallel composition operators '$|||$' (*interleaving composition*) and '$\|$' (*alphabetised parallel composition*). The former operator is used for the description of entirely independent, concurrently running processes; it interleaves all individual behaviours. The latter operator allows for the specification of a set of common events which all participating processes synchronise on. Synchronisation on common events can describe the synchronous execution of actions, but it can also be used to describe communication, as shown in the following example:

$$P = e!n : \mathbb{N} \rightarrow P \tag{2.11}$$

$$Q = e?n : \mathbb{N} \rightarrow Q \tag{2.12}$$

Here, event e acts as a communication channel on which values can be transferred between processes as described above. The communication direction is determined by the type of choice in the respective process. Here, P makes an internal choice about which natural number to send over channel e. Q is able to engage in any event in which a natural number is sent to it on channel e which corresponds with an external choice. Process P is thus able to send the value to Q. In order to describe their interaction, they need to be composed in parallel. This can be done as follows:

$$R = P \underset{e}{\parallel} Q \tag{2.13}$$

Here, process R describes the composite behaviour of processes P and Q which synchronise on event e. In this way, it is possible to describe complex temporal behaviours in a concise and elegant way.

Apart from its theoretical strengths, CSP also enjoys powerful tool support. FDR [210], a popular refinement checker, can be used to check specifications for deadlocks, livelocks or equivalences with respect to different semantic models such as traces, failures or divergences [211]. In order for a specification to be understandable by FDR, however, it needs to be re-written in CSP-M, a machine-readable version of CSP. Its syntax is similar to the syntax described above but it requires some of the mathematical operators to be replaced by conventional ASCII characters.

2.3.2.1 Operational semantics of CSP

An operational semantics of CSP can be described using *Labelled Transition Systems (LTS)* [210]. According to that, the LTS L_P describing the operational semantics of CSP process P can be defined as a tuple $L_P = (S, I, E, \rightarrow)$ where S denotes the set of states process P can be in, I is the set of initial states, $E = \alpha(P)$ is a set of events (the *alphabet* of P) and $\rightarrow \subseteq S \times E \times S$ is a transition relation such that $s \xrightarrow{e} s'$ if and only if $(s, e, s') \in \rightarrow$. The edges in the resulting state transition graph are labelled with event names. Given the possible nondeterminism of CSP processes, a state can have multiple outgoing transitions which are labelled with the same event.

Due to the event-based nature of CSP, states are typically not labelled when deriving an LTS from a process description; it is, however, possible — and often useful (especially in the context of model checking) — to do so and to label both states and transitions in the resulting model.

A CSP process can be easily interpreted operationally by traversing the LTS step-by-step, starting with the initial state. In this way, the CSP model can be used as a *path generator*, i.e. individual traces or paths through the LTS can be obtained without ever having to construct the full LTS (which might be infeasible). One can understand this operational interpretation as the *execution* of a CSP process.

2.3.3 Integrating Object-Z and CSP

Z and Object-Z have traditionally been used for the specification of state-based systems. Process algebras such as CSP [125] or CCS [189], on the other hand, have been particularly useful for the specification and verification of concurrent systems. Both formalisms have their respective strengths and weaknesses but with state modelling on one side and process interaction modelling on the other side, they can be seen as complementing each other quite well — particularly for the formalisation of complex, concurrent systems. It is therefore natural to think about potential combinations within a common formal framework. Consequently, several attempts to combine Z/Object-Z and CSP have been made in the past [94, 225, 227, 171]. Among the various approaches, we focus on the work of Smith and Derrick according to which Object-Z operations are translated into CSP events by interpreting the operation name as a communication channel and the value of each argument as a value communicated on the channel [227]. Similar to Z, operation parameters in Object-Z can be *decorated* in order to denote their 'direction': output parameters are decorated with a question mark (e.g. $p?$) and input parameters with an exclamation mark (e.g. $p!$). In order to associate the output of one operation with the input of another, we need to omit the decoration and refer to the *base name* of the argument, i.e. its undecorated description. To this end, a helper function β is introduced which removes the decoration from the argument, i.e. $\beta(p?) = \beta(p!) = p$. The following function $event$ describes the translation:

$$event((n, p)) = n.\beta(p) \tag{2.14}$$

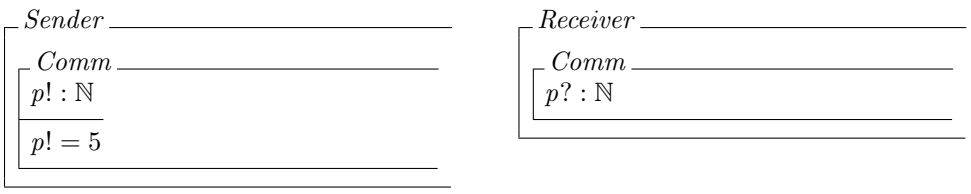
According to that, an Object-Z operation with name $n \in Id$ and parameters $p \in (Id \pfun Value)$ (as described in Section 2.3.1) is translated into the CSP event $n.\beta(p)$. An Object-Z class C can then be modelled as a parametrised CSP process C_i where i is an assignment of values to a subset of the state

of C which satisfies the initial conditions. For clarity, we use notation $C(i)$ instead. Let S denote the set of states and O denote the set of operations of class C. Then,

$$\alpha(C(i)) = \{event(op) \mid op \in O\} \tag{2.15}$$

denotes the *alphabet* of C which is defined as a set of events corresponding to the operations in O.

Example 1. Consider the following two class schemas:



Both classes comprise a $Comm$ operation. In $Sender$, this operation produces a natural number output (in this case '5'). In $Receiver$, the operation accepts a natural number as input. Given the description above, we can now translate the operations into events as follows:

$$event((Comm, p!)) = Comm.\beta(p!) = Comm.p \tag{2.16}$$

$$event((Comm, p?)) = Comm.\beta(p?) = Comm.p \tag{2.17}$$

Both resulting processes are able to communicate. Their alphabets are defined as follows:

$$\alpha(Sender) = \{Comm.p\} \tag{2.18}$$

$$\alpha(Receiver) = \{Comm.p\} \tag{2.19}$$

We can now describe the two classes as CSP processes, build their parallel composition and synchronise them on the $Comm$ operation:

$$Sender = Comm.5 \rightarrow Sender \tag{2.20}$$

$$Receiver = Comm.p \rightarrow Receiver \tag{2.21}$$

$$SendAndRecv = Sender \underset{Comm}{\parallel} Receiver \tag{2.22}$$

The first line describes the $Sender$ process which is always able to send value 5 on the $Comm$ channel. The second line describes process $Receiver$ which is always able to receive a natural number on the $Comm$ channel. We can thus build the parallel composition of the two processes (as described in Line 3) and synchronise them on the common $Comm$ event.

The integration of Object-Z operations with CSP processes as described above requires the usage of a restricted subset of the Object-Z language [224, 225]. The restrictions are as follows:

- Object instantiation and all related concepts such as polymorphism, class union, object containment are not allowed
- Operators used to model object interactions ('$\|$', '$\bullet$') are not allowed
- State hiding is required; Objects are not allowed to access the state of other objects directly, only through their respective procedural interface
- Usage of the sequential composition operator ('$\fatsemi$') is not allowed

Given the translation of Object-Z classes and operations into CSP processes and events, the parallel composition of two (equally-named) operation schemas belonging to two different classes using the Object-Z operator '$\|$' is then semantically equivalent to the interpretation of the two classes as CSP processes and their synchronous parallel composition using the CSP operator '$\|$'. This makes it possible to use Object-Z's strong state modelling capabilities to formalise the internals of a component and CSP's elegant process interaction modelling capabilities to describe a population of concurrently acting components. This aspect is particularly appealing against the background of complex systems such as agent-based simulations since it allows developers to use Object-Z for the modelling of agents and agent internals and CSP for the modelling of the population. This approach will be followed in the description of the formal framework in Chapter 3.

2.4 Verification

This section describes verification approaches which are related to the work described in this thesis. We start with a brief description of finite state automata in Section 2.4.1, followed by a description of Linear Temporal Logic (LTL) in Section 2.4.2, an overview of model checking and its probabilistic and statistical variants in Section 2.4.3, and runtime verification in Section 2.4.4.

2.4.1 Finite state automata

In the context of formal verification, the behaviour of the system under study is typically represented as some kind of *finite state automaton (FSA)* which can be seen as a finite variant of the transition system introduced above. Formally, a FSA $\mathcal{M} = (S, I, \rightarrow, AP, L)$ is a 5-tuple where S is a finite set of states, $I \subseteq S$ is a set of initial states, $\rightarrow \subseteq S \times S$ is a transition relation, AP is a set of atomic propositions and $L : S \rightarrow 2^{AP}$ is a labelling function that labels each state with a set of atomic propositions that are true in this state. $s \rightarrow s'$ is often written as a shortcut for $(s, s') \in \rightarrow$. A $path$ or $trace$ of $\mathcal{M}$ is then defined as a sequence of states $\langle s_0, s_1, s_2, ..., \rangle$, starting in an initial state.

2.4.2 Linear temporal logic (LTL)

Temporal logics can be roughly subdivided into two general classes: *branching time logic*, examples of which are CTL and CTL* and *linear temporal logic* (LTL) [57]. The former class assumes that there is always a choice between different potential successor states at each time step and thus views time as an exponentially growing tree of 'possible worlds' in which the future is not pre-determined. In contrast, linear time logic views time as a linear sequence of states. The differences between branching time and linear time logics in terms of expressiveness have been discussed extensively in the literature and shall not be elaborated upon here. Interested readers may, for example, refer to [62].

2.4.2.1 Syntax and semantics

The approach described in this report is based upon the analysis of individual paths. Since each path describes a sequence of states, the notion of linear time is used inherently. Below, we describe the syntax and semantics of LTL and give a satisfaction relation with respect to a simple sequence of states. The syntax of an LTL formula ϕ is defined by the following grammar:

$$\phi ::= \text{true} \mid a \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg \phi \mid \mathbf{X}\phi \mid \phi \, \mathbf{U} \, \phi \tag{2.23}$$

LTL formulae are all path formulae. The basic building blocks are atomic propositions $a : AP$, the Boolean connectives $\wedge$ ('and') and $\vee$ ('or') and the temporal connectives $\mathbf{X}$ ('next') and $\mathbf{U}$ ('until') included the bounded variant. For LTL formula ϕ, $\mathbf{X}\phi$ holds in a state s (denoted $s \models \phi$) if ϕ holds in its direct successor state. For LTL formulae ϕ_1 and ϕ_2, $\phi_1 \, \mathbf{U} \, \phi_2$ holds in state s if ϕ_1 holds in s and

continues to hold until ϕ_2 becomes true. In some cases, LTL is extended with a *bounded until operator* $\mathbf{U}^{\leq k}$; here, $\phi_1 \mathbf{U}^{\leq k} \phi_2$ holds in state s if ϕ_1 holds in s and ϕ_2 holds at some point within the next k time steps. Other logical connectives such as '$\Rightarrow$' or '$\Leftrightarrow$' can be derived in the usual manner:

$$\phi_1 \Rightarrow \phi_2 \equiv \neg\, \phi_1 \vee \phi_2 \tag{2.24}$$

$$\phi_1 \Leftrightarrow \phi_2 \equiv (\phi_1 \Rightarrow \phi_2) \wedge (\phi_2 \Rightarrow \phi_1) \tag{2.25}$$

Furthermore, additional temporal connectives $\mathbf{F}$ ('finally','eventually'), $\mathbf{G}$ ('globally','always'), $\mathbf{W}$ (a weak version of the 'until' operator[2]) and $\mathbf{R}$ ('release') can be derived as follows:

$$\mathbf{F}\phi \equiv \text{true } \mathbf{U}\, \phi \tag{2.26}$$

$$\mathbf{G}\phi \equiv \neg\, \mathbf{F}(\neg\, \phi) \tag{2.27}$$

$$\phi_1 \mathbf{W} \phi_2 \equiv (\phi_1 \,\mathbf{U}\, \phi_2) \vee \mathbf{G}\phi_1 \tag{2.28}$$

$$\phi_1 \,\mathbf{R}\, \phi_2 \equiv \neg\, (\neg\, \phi_1 \,\mathbf{U}\, \neg\, \phi_2) \tag{2.29}$$

In order to describe the 'meaning', i.e. the *semantics*, of an LTL property with respect to sequences of states, a *satisfaction relation* can be defined as follows.

Let $\sigma = \langle s_0, s_1, s_2, ..., \rangle$ denote an infinite trace. The satisfaction of an LTL formula ϕ can then be defined as follows:

$$\sigma \models a \Leftrightarrow a \in L(\sigma[0]) \tag{2.30}$$

$$\sigma \models \phi \Leftrightarrow \sigma[0] \models \phi \tag{2.31}$$

$$\sigma \models \neg\, \phi \Leftrightarrow \sigma[0] \not\models \phi \tag{2.32}$$

$$\sigma \models \phi_1 \wedge \phi_2 \Leftrightarrow \sigma[0] \models \phi_1 \wedge \sigma[0] \models \phi_2 \tag{2.33}$$

$$\sigma \models \mathbf{X}\phi \Leftrightarrow \sigma[1..] \models \phi \wedge |\, \sigma \,| > 0 \tag{2.34}$$

$$\sigma \models \phi_1 \,\mathbf{U}\, \phi_2 \Leftrightarrow \exists j \geq 0 \text{ s.t. } \sigma[j..] \models \phi_2 \wedge \forall\, 0 \leq i < j, \sigma[i..] \models \phi_1 \tag{2.35}$$

The semantics of the derived operators can be given accordingly:

[2] As opposed to the conventional 'until' operator, the weak variant does not require ϕ_2 to become true at some point in the future.

$$\sigma \models \mathbf{F}\phi \Leftrightarrow \exists\, i \geq 0 \text{ s.t. } \sigma[i..] \models \phi \tag{2.36}$$

$$\sigma \models \mathbf{G}\phi \Leftrightarrow \forall\, i \geq 0, \sigma[i..] \models \phi \tag{2.37}$$

$$\sigma \models \phi_1 \mathbf{W} \phi_2 \Leftrightarrow \exists j \geq 0 \text{ s.t. } \sigma[j..] \models \phi_2 \wedge \forall\, 0 \leq i < j, \sigma[i..] \models \phi_1$$
$$\textbf{or } \forall\, i \geq 0, \sigma[i..] \models \phi_1 \tag{2.38}$$

In addition to atomic operators, interesting combinations of operators (*dual operators*) can also be defined. For example, $\mathbf{FG}\phi$ states that "ϕ will eventually hold forever". $\mathbf{GF}\phi$ states that "ϕ is satisfied infinitely often".

It is important to emphasise that the semantics of LTL are defined over *infinite traces*. This has important implications on the verification of properties upon simulation traces which are by definition finite. This issue is discussed in further detail in Section 5.2.2.

2.4.2.2 LTL with constraints

In its basic form described above, LTL comprises atomic propositions and temporal and Boolean operators. From a practical point of view, it is convenient to also allow for the specification of simple arithmetic operations directly in the temporal logic. The evaluation of expressions like '$numPurchases > x'$ where $numPurchases$ and x are numeric values then becomes an inherent part of the model checking process.

The integration of numeric functions and simple arithmetic expressions has already been explored and resulted in the definition of *LTLc (LTL with constraints)* [76, 80]. Its usefulness for the verification of complex quantitative and qualitative properties involving external data in the domain of biochemical systems has been shown, for example, by Fages and Rizk [90]. The grammar of LTLc is given below:

$$\phi ::= \text{true} \mid a \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\, \phi \mid \mathbf{X}\phi \mid \phi\, \mathbf{U}\, \phi \mid \phi\, \mathbf{U}^{\trianglelefteq k}\, \phi \mid val \trianglelefteq val \tag{2.39}$$

where $\trianglelefteq \in \{>, <, \geq, \leq, =, \neq\}$ and val can be either an arithmetic expression, the result of a numeric function or a real number:

$$val ::= val \oplus val \mid val^{val} \mid func \mid \mathbb{R} \tag{2.40}$$

where $\oplus \in \{+, -, *, /\}$. In terms of satisfaction, the LTL fragment of LTLc is dealt with in the same way as in conventional LTL and will thus be omitted. For the remaining fragment, let $Eval : \sigma \times val \rightarrow \mathbb{R}$ be an evaluation function which accepts any state s and any val as an input, evaluates val on s and returns a real number. Then the satisfaction of $val \trianglelefteq val$ can be given as follows:

$$\sigma \models val_1 \trianglelefteq val_2 \Leftrightarrow Eval(\sigma[0], val_1) \trianglelefteq Eval(\sigma[0], val_2) \tag{2.41}$$

2.4.3 Model checking

Model checking [61] is a powerful and popular verification technique which uses a formal representation $\mathcal{M}$ of the system under consideration (e.g. a finite-state representation) together with a specification of the system's desired properties p, typically given in some kind of temporal logic (e.g. CTL, CTL* or LTL). The verification of a system's correctness is then done by checking whether a given property holds in the model (formally $\mathcal{M} \models p$) by examining all possible execution paths. In the case of violation, the model checker can provide a counterexample. Verification criteria can be classified into different categories as described in Section 4.3. This section introduces the general characteristics of model checking as well as its probabilistic and statistical variants.

2.4.3.1 General characteristics

In the context of model checking, it is important to distinguish between the verification of branching time (e.g. CTL, CTL*) and linear temporal logic (LTL). In the former case, model checking typically consists of a recursive labelling procedure in which each state in the model $\mathcal{M}$ is labelled with all those subformulae of the property ϕ to be verified (including ϕ itself) which hold in this state. The overall model satisfies property ϕ (denoted $\mathcal{M} \models \phi$) precisely if the initial state of $\mathcal{M}$ is labelled with ϕ. In the case of LTL, a property ϕ (which is defined over *traces* rather than over a tree, as described in Section 2.4.2) defines a ω-language $\mathcal{L}(\phi)$ over alphabet 2^{AP}, i.e. a set of infinite sequences of words, where each word is a finite sequence of atomic propositions which are allowed to hold in the respective state. Model checking an LTL property ϕ against a model $\mathcal{M}$ then amounts to determining whether all

executions of $\mathcal{M}$ are contained in $\mathcal{L}(\phi)$ which can be achieved in an automata-theoretic way. To this end, a *Büchi automaton* of the negation of property ϕ is constructed (denoted $\mathcal{A}_{\neg\,\phi}$). Intuitively, $\mathcal{A}_{\neg\,\phi}$ describes all executions which do *not* satisfy ϕ. It is then checked whether the product of $\mathcal{M}$ and $\mathcal{A}_{\neg\,\phi}$ (which is denoted $\mathcal{M} \otimes \mathcal{A}_{\neg\,\phi}$ and describes all those behaviours of $\mathcal{M}$ which do *not* satisfy ϕ) is empty, in which case the property is satisfied, i.e. we have $\mathcal{M} \models \phi \Leftrightarrow \mathcal{M} \otimes \mathcal{A}_{\neg\,\phi} = \varnothing$. A good overview of automata-theoretic model checking is given by Vardi [232].

Model checking can be roughly subdivided into *explicit-state model checking* and *symbolic model checking* [181]. In the former case, an explicit description of the underlying transition matrix is used throughout the verification process (as described above). Examples of explicit-state model checkers are *SPIN* [128], *Bandera* [64], and *Java Path Finder* [1]. Symbolic model checking uses a different approach and encodes a system state into a formula in propositional logic (typically a Boolean formula). The full graph of possible execution paths can then be represented with a compact data structure, for example a *Binary Decision Diagram (BDD)* [181]. Due to the much more compressed representation, symbolic model checking approaches are typically more efficient than their explicit-state counterparts. With symbolic model checking, it is possible to include hundreds of state variables into the verification process and to check systems with up to 10^{120} states [60, 41]. Examples of symbolic model checkers are *NuSMV* [56], *PRISM* [123] and *MCMAS* [163].

Despite impressive advances, however, exponential growth of the underlying finite-state model (the so-called state space explosion) remains a central problem which makes the verification of non-trivial real-world systems difficult or even impossible. Apart from symbolic model checking, a number of techniques has been developed in order to tackle this problem, such as

1. reduction (e.g. symmetry reduction [59], partial order reduction [61]);
2. approximation (e.g. through sampling) [117];
3. composition (e.g. assume-guarantee reasoning [149]);
4. SAT-based model checking [58];
5. abstraction (e.g. counting abstraction [92]); and
6. bounded model checking [31].

The question of which technique is most appropriate is highly dependent upon the nature of the underlying system and cannot be answered in general. Symmetry reduction can be helpful if the system is

mostly homogeneous in nature and consists of multiple similar components; when dealing with large-scale systems that comprise an unmanageably large number of states, abstraction and approximation techniques have shown to produce promising results. However, both approximation and abstraction involves information loss which needs to be taken into account when interpreting the verification results. Depending on the safety requirements of the underlying system, potential concessions to the verification strength need to be evaluated critically.

Since this work is largely based on the idea of sampling-based, approximate model checking, this approach is described in more detail in Section 2.4.3.3 below.

2.4.3.2 Probabilistic model checking

The classical variant of model checking assumes that transitions in the underlying FSA are deterministic and non-probabilistic. As a consequence, verification questions are purely qualitative in nature and can thus be answered with either *'yes'* or *'no'*. In many cases, however, state transitions need to be stochastic in order to reproduce the underlying system's dynamics with a sufficient degree of realism (e.g. when looking at randomised algorithms or systems with inherent uncertainty). In this case, verification is slightly more complex. Instead of providing clear yes/no answers, probabilistic verification returns the probability of an event happening, e.g. "what is the probability of finally reaching state s?".

In order to cope with stochastic systems, a probabilistic variant of model checking [148] has been developed. It uses a Markov model — typically a discrete- or continuous-time Markov chain (DTMC/CTMC) or a Markov Decision Process (MDP) — as the underlying formal model. Properties are specified in a probabilistic temporal logic, for example PCTL [114]. Instead of universal and existential quantification as in CTL, PCTL uses a probabilistic operator $\mathbb{P}_{\bowtie p}(\varphi)$, where $p \in [0,1]$, $\bowtie \in <, >, \leq, \geq$ and φ is a path formula. A PCTL expression states that "φ is true with probability $\bowtie p$". An example of a probabilistic model checker is PRISM [148].

2.4.3.3 Statistical model checking

Conventional model checking aims to find an accurate solution to a given property by exhaustively searching the state space of the underlying FSA. As described above, this is only possible in cases where the state space is of manageable size. One solution to this problem which works particularly well for probabilistic systems such as the ones described in the previous section is to use a *sampling approach*

and employ statistical techniques in order to generalise the so obtained results to the overall state space. In this case, a number of traces are sampled from the underlying state space and the property is checked on each of them. Techniques for statistical inference such as, for example, *hypothesis testing*, can then be used to determine the significance of the results. Approaches of this kind are summarised under the umbrella of *statistical model checking*; a good overview has been given by Legay and Delahaye [153]. Due to its independence from the underlying state space, statistical model checking allows for the verification of large-scale (or even infinite) systems in a timely, yet approximate manner.

In this context, it is useful to distinguish between the verification of *black box* and *white box systems*. A black box system is a system whose execution is not under the control of the person performing the verification experiment (e.g. because its repeated execution is prevented by resource constraints). In this case, inferences can only be drawn from output of the system which has been produced at some point in the past. This is in contrast to a white box system which a user has full control of and which allows for repeated simulation and a custom number of outputs. In this case, different levels of accuracy can be achieved by varying the number of paths that a property is being verified upon.

For the stochastic verification of white box systems by means of Monte Carlo simulation, a range of approaches have been proposed in the literature [117, 112, 80]. We focus here on the idea of *Approximate Probabilistic Model Checking (APMC)* proposed by Hérault and Lassaigne which allows for the verification of linear time properties with a clearly quantifiable level of accuracy and confidence using a *fixed* number of replications [117]. A critical advantage of this approach is that it decouples the number of paths to be evaluated from the size of the underlying system and thus provides a high level of scalability.

Algorithm 1 Generic Approximation Algorithm $\mathcal{GAA}$

Require: $pathGen, \phi, k, \epsilon, \delta$
  $N := \ln\left(\frac{2}{\delta}\right) \cdot \frac{1}{2\epsilon^2}$
  $A := 0$
  for $i := 1$ to N **do**
    1. Generate a random path σ of length k using $pathGen$
    2. If ϕ is true on σ then $A := A + 1$
  end for
  return A/N

APMC is based on the following principle: n paths from the state space underlying the target system $\mathcal{M}$ are obtained through random sampling. It is then checked for each path σ whether σ satisfies a given property ϕ, denoted $\sigma \models \phi$. Let po denote the number of positive outcomes. The overall probability of $\mathcal{M}$ satisfying ϕ can then be approximated by $Pr(\mathcal{M} \models \phi) = \frac{po}{n}$.

The confidence in the results is determined as follows. The verification of ϕ on path σ can be considered a Bernoulli experiment with either positive or negative outcome. The evaluation of a property on a set of paths thus yields a *Binomial distribution* which, for sufficiently large values of n can be reasonably approximated by a normal distribution. In 1963, Wassily Hoeffding proved an upper bound on the probability of the sum of random variables deviating from its expected value [127]. According to his idea (which became known as the *Hoeffding inequality*), at least $\ln\left(\frac{2}{\delta}\right)/2\epsilon^2$ samples need to be obtained in order to achieve a result Y that deviates from the real probability X by at most ϵ with probability at least $1-\delta$, i.e. $Pr(|X-Y| \leq \epsilon) \geq 1-\delta$.

This idea forms the basis for the *Generic Approximation Algorithm ($\mathcal{GAA}$)* presented by Hérault and Lassaigne and outlined in Algorithm 1 [117]. It accepts four inputs: a path generator $pathGen$[3], an LTL property ϕ, the desired path length k, an approximation parameter ϵ and a confidence parameter δ. The algorithm obtains N sample paths where $N = \ln\left(\frac{2}{\delta}\right)/2\epsilon^2$ is a function of ϵ and δ. Property ϕ is evaluated separately for each path. Each time a path satisfies ϕ, a counter variable A is incremented. The overall probability of ϕ is then estimated as A/N.

A useful benefit of the Hoeffding bounds is that a high level of confidence can be achieved rather cheaply. For example, in order to achieve 95% confidence with an accuracy of $\pm 5\%$, 738 sample paths need to be analysed. In order to achieve 99% confidence with the same level of accuracy, 1,060 sample paths need to be analysed. Achieving a high level of accuracy, on the other hand, is costly. For example, in order to achieve $\pm 1\%$ accuracy and 95% confidence, we already need 18,445 sample paths! A comparison between the number of paths needed to achieve a certain level of accuracy and the number of paths needed to achieve a certain level of confidence is shown in Figure 2.2.

Due to its focus on individual paths and its independence of the size of the underlying state space, APMC represents an interesting technique for the verification and validation of large-scale agent-based simulations. It forms the basis of the runtime verification approach developed as part of this work. In Chapter 6, we describe a modified version of Algorithm 1. It uses a different (algorithmic) sample size determination mechanism which is based on the application of the *Binomial theorem* and achieves a significant reduction in the total sample size necessary for obtaining verification results with a certain level of accuracy and confidence.

[3] In the original paper, this parameter is referred to as a '$diagram$' [117].

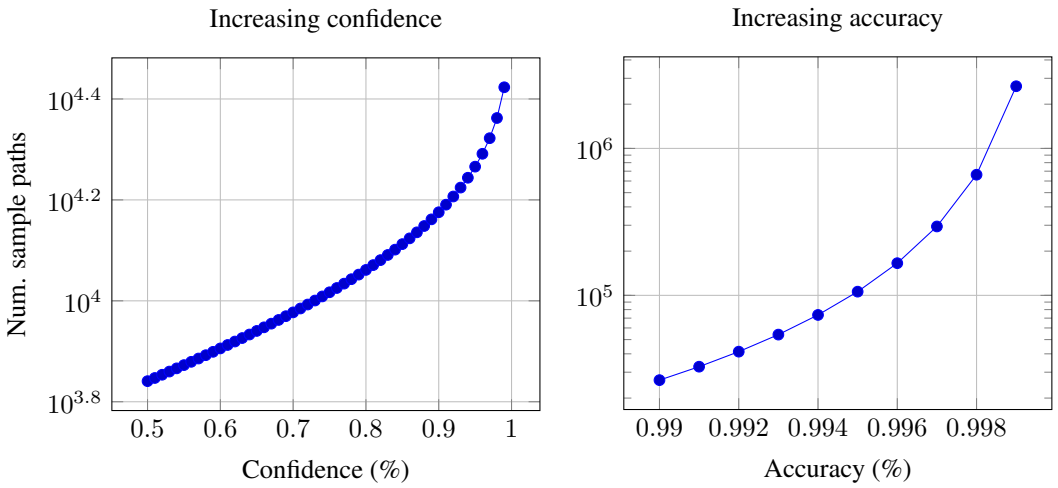


FIGURE 2.2: Number of sample paths necessary to achieve a desired level of confidence (left) and accuracy (right) using Hoeffding bounds

2.4.4 Runtime verification

As described above, the application of conventional model checking is severely constrained by the size of the underlying system. *Runtime verification* attempts to circumvent this problem by focussing on the *current execution* of a system instead of its universal behaviour [155]. In that respect, runtime verification bears a strong similarity with testing. However, in contrast to conventional testing, runtime verification typically allows for the formulation of the system's desired behaviour in a less ad-hoc and more rigorous, e.g. logic-based, way and can thus be considered more formal.

Runtime verification represents a large research area in its own right and we can only give a very superficial description here. A typical runtime verification approach has the following three characteristics:

1. The behaviour of the system under consideration is observed while the system is running
2. A property describing the system's desired behaviour is checked against the current execution trace
3. A result is reported as soon as the property has been satisfied (or violated)

Due to its focus on individual execution traces, runtime verification views time as a linear flow and properties are thus often formulated in a variant of LTL. Those properties are then translated into a

monitor which is used to observe the execution of the system and report any satisfaction or violation that may occur. In order for a monitoring approach to be efficient, it necessarily needs to be *forward-oriented*; having to 'rewind' the execution of a system in order to determine the truth of a property is generally not an option. In terms of monitor construction, two different approaches — *automaton-based* and *symbolic* — can be distinguished. They are briefly described below.

The automaton-based approach is related with automata-theoretic model checking of LTL properties described in Section 2.4.3 and involves the construction of an automaton, for example a finite state automaton, a Büchi automaton or an alternating finite automaton. It represents the language $\mathcal{L}(\phi)$ defined by a given LTL property ϕ which describes all allowed traces of the system. One typical problem in runtime verification is that the truth of a property ϕ at time t may need additional information about the future behaviour of the trace which is not yet available. From the monitor's point of view at a particular point in time, the future is not foreseeable. Depending on the type of automaton being used, transitions may thus need to be taken nondeterministically which raises questions about the correct *traversal mode*. In the case of depth-first traversal, the automaton can only ever be in a single state, yet it may require backtracking if the 'wrong' path has been taken. In the case of breadth-first traversal, the automaton can be in a number of states at the same time, yet the number of currently active states may grow exponentially over time. An example of a nondeterministic Büchi automaton for the (slightly contrived) LTL formula '$(a \wedge \mathbf{X}b) \vee (a \wedge \mathbf{X}c)$' is shown in Figure 2.3. The automaton contains only one initial (state 0) and one accepting state (state 4). Edge labels denote properties that need to hold in order for the automaton to transition into the next state. The graph shows that, initially, a is required to hold in order for the entire formula not to be violated, i.e. in order to transition into a successor state. There are, however, two different successor states (1 and 3) which satisfaction of a might lead to, a choice of which needs to be made nondeterministically. A depth-first traversal approach would, for example, transition into state 1 at time 0 and then, if b does not hold at time 1, backtrack to state 0 and then transition into state 3 in order to check whether c holds. A breadth-first traversal approach would, at time 0, keep both states 1 and 3 as possible options in memory and decide based on the respective truth of b or c which path to discard (if any). Alternating finite automata (AFA), on the other hand, provide a solution by offering both existential as well as universally quantified transitions. In the case of uncertainty about the correct future states from the point of view of a particular state s, an AFA may allow for the transitioning into *all* possible successor states of s.

The description of automaton construction techniques is beyond the scope of this work; detailed information can be found in the relevant literature [102, 99].

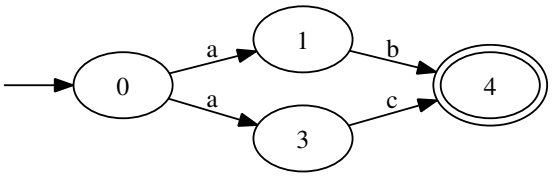


FIGURE 2.3: A nondeterministic Büchi automaton representing a LTL formula

The second, symbolic approach is based on the idea of *formula manipulation*. Instead of constructing a full automaton as mentioned above, the monitor is represented as a formula which describes (i) the requirements that the currently observed state needs to satisfy in order to either satisfy (or at least not violate) a given property ϕ, and (ii) those requirements which the rest of the trace needs to fulfil in order for ϕ to be eventually satisfied (so-called *obligations*). The symbolic approach makes use of the recursive nature of the temporal logic by exploiting *expansion laws*, the ones for LTL of which are shown in Table 2.3[4]. It is easy to see that, according to them, formula '$(a \wedge \mathbf{X}b) \vee (a \wedge \mathbf{X}c)$' can be adopted without any changes and used as a monitor right away.

In this context, it is important to mention Pnueli's work in which he contrasts *acceptors* (i.e. automata-based monitors) with *temporal testers* (symbolic monitors) [202]. A temporal tester can be described as a monitoring process which observes a trace against the background of given correctness criteria (e.g. formulated in linear temporal logic) and reports for each state in the trace whether the given formula has already been satisfied or violated. As the name suggests, temporal testers are closely related with the idea of testing described above. However, due to their reliance upon properties formulated in temporal logic, they offer a higher level of rigour than conventional testing. Pnueli also describes several advantages of the tester over the acceptor approach. As opposed to an acceptor (e.g. a Büchi automaton) typically used in LTL model checking, a tester for a formula ϕ is able to decide not only the containment of *infinite* input sequences in the language defined by ϕ, but also the containment of *any finite suffix* which is particularly important for the idea of runtime verification. Furthermore, as opposed to an acceptor, a temporal tester for a formula ϕ is *compositional*, i.e. it can be assembled from the testers of the subformulae of ϕ. This means that, for any logic-based language, temporal testers only need to be specified for the atomic constituents of the language (e.g. '$\mathbf{X}$' and '$\mathbf{U}$' in the case of LTL). Testers for more complex formulae can then be constructed easily by conjoining or disjoining the testers for the individual subformulae. Given the automaton-based monitors for two LTL formulae ϕ_1 and ϕ_2, on the other hand, it is not straightforward to see how a monitor for $\phi_1 \wedge \phi_2$ can be constructed by simply

[4] Although 2.45 and 2.46 can be derived from 2.43 and 2.44, it is handy to define them separately due to their frequent occurrence

$$\mathbf{X}\phi \equiv \text{true} \wedge \mathbf{X}\phi \tag{2.42}$$
$$\phi_1 \,\mathbf{U}\, \phi_2 \equiv \phi_2 \vee (\phi_1 \wedge \mathbf{X}(\phi_1 \,\mathbf{U}\, \phi_2) \tag{2.43}$$
$$\phi_1 \,\mathbf{R}\, \phi_2 \equiv \phi_2 \wedge (\phi_1 \vee \mathbf{X}(\phi_1 \,\mathbf{R}\, \phi_2) \tag{2.44}$$
$$\mathbf{F}\phi \equiv \phi \vee \mathbf{XF}\phi \tag{2.45}$$
$$\mathbf{G}\phi \equiv \phi \wedge \mathbf{XG}\phi \tag{2.46}$$

TABLE 2.3: Expansion laws for some LTL operators

combining their individual automata. Compositionality is particularly helpful if extensions of existing logics are to be developed. One important aspect of this work is the definition of a LTL-based specification language for temporal behaviours of large-scale populations of agents. The ability to construct monitors for new operators by simply combining existing ones is very helpful in this case and shall be used extensively for the construction of verification algorithms described in Chapter 6. A temporal tester is further called *optimal* if it is able to extract as much information from the trace as possible and is thus able to report any violation as soon as it occurs [201]. Temporal testers have been implemented for a variety of logics and logic-based languages, e.g. LTL, PSL or MITL [202].

2.5 Verification of multiagent systems and simulations

In this section, we give a brief overview of verification approaches for both non-probabilistic and probabilistic multiagent systems, followed by an overview of existing work on simulation verification. In alignment with the focus of this work, we restrict the description of verification approaches for general multiagent systems (Sections 2.5.1 and 2.5.2) to model checking-based verification and exclude approaches based on testing[5], debugging or theorem proving. Due to the scarcity of existing work, we relax this restriction in the case of simulations and also include monitoring- and testing-based approaches (Section 2.5.3).

[5]Despite the conceptual proximity of testing and runtime verification, existing work on testing multiagent systems focusses on *distributed* systems and is thus not directly relevant here [100].

2.5.1 Model checking general multiagent systems

Since its beginnings around 25 years ago, model checking has gained huge significance in computer science and software engineering in particular and has been successfully applied to many real-world problems. Due to the availability of model checking tools and their applicability to practical problems, significant breakthroughs in terms of software correctness could be achieved[6]. Model checking has also gained increasing importance in the multiagent community and numerous approaches have been presented in literature. Since the focus of this work is on the verification of agent-based *simulations* which differ from general multiagent systems in a range of characteristics, we restrict the following description to three seminal and influential model checking approaches.

Instead of proposing a verification approach on top of existing agent frameworks or languages, Wooldridge *et al.* presented MABLE, a programming language whose compiler natively supports verification by model checking temporal and epistemic properties of the underlying system [244]. In terms of verification, MABLE makes use of the SPIN model checker, a freely available model checking system [128]. Using SPIN, the desired dynamic properties of a system can be specified in propositional LTL and checked against the execution of an abstract model specified in Promela, SPIN's specification language. The MABLE compiler translates the program into Promela code and the *claims* (the properties to be checked) into LTL which makes them compatible with the SPIN model checking system. In this way, temporal claims about the system's behaviour can be combined with the source code and checked automatically at compile time. If the verification fails, the MABLE compiler provides a counterexample which helps the developer to locate the cause of the problem. This approach is attractive because it makes the verification of temporal properties an inherent feature of the underlying programming language. Furthermore, by automating the transformation of the program code into a Promela model, it relieves the developer from manually creating a finite-state model of the underlying application.

In 2006, Bordini *et al.* presented work on model checking BDI-based multiagent systems written in a finite subset of AgentSpeak(L), a well-known agent-based programming language [35, 33]. Since model checking can only operate on finite state spaces, the author's first step is to transform the full version of AgentSpeak(L) into a finite subset (which they call AgentSpeak(F)) by constraining the maximum size of data structures and communication channels. On top of that, they describe an approach to transform AgentSpeak(F) programs into Promela, the specification language of the SPIN model checker [128]. In addition to the SPIN-based approach, the paper also describes how AgentSpeak(F) programs can

[6] In 2001, SPIN was awarded the prestigious ACM Software Systems Award

be verified using Java Path Finder (JPF) [1]. The core of JPF is a Java Virtual Machine (JVM) which takes a program in bytecode format as input and runs once for each possible execution path (in contrast to the conventional JVM which runs a program only once in total). JPF has built-in default checks for deadlocks and unhandled exceptions and can be extended in any direction by providing additional Listeners which check for custom violations. Since JPF checks Java programs only, AgentSpeak(F) programs need to be transformed first. The authors describe an automated transformation approach from AgentSpeak(F) to Promela elsewhere [34]. For the formulation of desired properties, Bordini *et al.* propose a specification language which incorporates temporal and modal aspects and allows for the specification of properties with respect to agent-specific entities like beliefs, desires and intentions.In addition to checking AgentSpeak(F)-based programs, the authors have also developed Agent JPF (AJPF), a methodology and toolset which abstracts from the underlying programming language and allows for the verification of agent-based programs written in different languages [33]. Language independence is achieved by means of an *Agent Interpretation Layer (AIL)* which provides a basis for the construction of interpreters for different agent programming languages by encompassing their main concepts.

In contrast to the explicit model checking approaches described in the previous sections, Lomuscio *et al.* developed MCMAS, a symbolic model checker tailored to agent-based specifications and scenarios [163]. MCMAS supports the verification of conventional CTL, epistemic logic, alternating time logic and deontic logic using ordered binary decision diagrams on *interpreted systems*, a formalism for modelling distributed systems which was first presented by Halpern and Fagin [113]. In order to facilitate the verification process, MCMAS also provides a graphical user interface which is realised as an Eclipse plug-in and offers dynamic syntax checking, outline view, text formatting and syntax highlighting. The GUI also supports interactive execution by allowing the user to explore the model in both a symbolic and an explicit way. An interesting additional feature (which is also provided by MABLE described above) is the possibility to further analyse the results of the verification process by displaying both witnesses (executions in which a given dynamic property has been satisfied) and counterexamples (executions in which the property has been violated).

In order to allow for the verification of larger agent populations, model checking algorithms for temporal-epistemic properties have also been combined successfully with ideas such as bounded model checking [162], partial order reduction [161] and parallelisation [147]. Despite impressive advances, verification still remains limited to relatively small populations. A particularly promising approach which has been proposed recently is based on *parametrised interleaved interpreted systems (PIIS)* [143]. A PIIS is a

special type of interpreted systems which models a *template agent* from which all agents in the population are derived (i.e. agents are required to be identical) together with a parameter which denotes the number of agents. Actions represented within the template agent are either asynchronous or synchronous, the latter of which are performed jointly. Whilst the verification of parameterised systems is undecidable in general, Lomuscio and Kouvaros proposed a *cutoff technique* for a particular class of PIIS together with a parametrised temporal-epistemic logic [144]. Within this particular class of PIIS, the approach allows for the verification of populations with an unbounded number of agents.

In general, it can be said that most model checking approaches for general multiagent systems focus on rich properties (e.g. including epistemic or deontic modalities) and a comparatively low number of agents rather than on large populations. Furthermore, they do not allow for the explicit specification of randomness in the underlying multiagent system. Approaches which aim to address the latter problem are presented in the following section.

2.5.2 Verification of probabilistic multiagent systems

In recent years, probabilistic model checking has gained increasing importance for the verification of general multiagent systems. Some of these approaches are briefly described below.

An interesting approach to verify the emergent behaviour of robot swarms using probabilistic model checking has been presented by Konur *et al.* [141]. In order to tackle the combinatorial explosion of the state space, the authors exploit the high level of symmetry in the model and use *counter abstraction* [92]. In this case, instead of creating the parallel composition of the single agents' state machines, the system is represented with a single, system-level state machine. This global representation is similar to the individual state machines but contains an additional counter which determines how many individual agents are in the current state. In doing so, the authors manage to transform the originally exponential into a polynomial problem[7]. This is a significant improvement, however, since the resulting problem is still exponential in the number of agent states, the approach remains limited to small-scale systems.

Ballarini *et al.* [16] apply probabilistic model checking to a probabilistic variant of a negotiation game. They use PRISM [148], a probabilistic model checker, to verify PCTL-based properties addressing two aspects of the system: (i) the value at which an agreement between two agents bargaining over a single resource is reached, and (ii) the delay for reaching an agreement. In this scenario, the overall state space

[7] To be precise: The resulting problem is polynomial in the number of agents and exponential in the number of agent states.

is small and therefore combinatorial explosion is not an issue. According to the authors, probabilistic verification provides an interesting alternative to analytical and simulation methods and can provide further insight into the system's behaviour.

Dekhtyar *et al.* [71] assume that randomness in a multiagent system can be due to (i) uncertainty of communication channels and (ii) uncertainty of action. In their paper, they describe a method to translate a multiagent system into a finite-state Markov chain and analyse the complexity of probabilistic model checking of its dynamic properties. Apart from mentioning the exponential complexity of both state space creation and verification, however, the authors do not present any ways to circumvent this problem. The verification of epistemic properties has also been addressed against the background of probabilistic agent-based systems.

Wan *et al.* [235] describe a model checking approach for probabilistic multiagent systems on Discrete-Time Markov Chains (DTMCs). In order to cope with transition probabilities, the formalism of Interpreted Systems is extended with probability attributes and so associated with the DTMC. For the formulation of properties, the authors propose PCTLK, an epistemic, probabilistic branching-time logic which extends CTL with probabilistic and epistemic operators. In their paper, the focus of interest is on the verification of epistemic properties and complexity issues are not being addressed.

Another interesting approach for nondeterministic multiagent systems based on the idea of *formally guided simulation* has been proposed by Salem da Silva in his PhD thesis [213]. Here, the behaviour of individual agents as well as of the environment is formulated in a simple specification language which is based on the π-calculus. The behaviour of the whole system $\mathcal{M}$ is defined by the operational semantics of the π-calculus. The desired (or undesired) behaviours of the system form the *simulation purposes* $\mathcal{SP}$ which are formulated in the same specification language; they are used to guide the simulation of the system. Instead of focussing on agent internals, Salem da Silva takes a purely *behavioural* approach and describes agents as simple black box entities. The 'correct fragment' of $\mathcal{M}$ is described by the product automaton $\mathcal{M} \otimes \mathcal{SP}$[8] which is constructed on-the-fly in order to guide the simulation. The approach avoids exhaustive state space exploration, yet it suffers from a number of limitations. First, π-calculus descriptions can grow very large which restricts the technique to fairly simple systems with few components. The approach also does not currently take into account transition probabilities and thus cannot make any statements about the probabilities of events. And finally, in cases where the state space fragment defined by the simulation purpose is large, the approach still suffers from combinatorial explosion.

[8] This corresponds with the idea of automaton-based model checking described in Section 2.4.3 [232].

Whereas the approaches described above (except the last one) are able to deal with randomness, they do not address the problem of scalability and remain limited to relatively small-scale systems. They are thus not directly applicable to the verification of typical agent-based simulations.

2.5.3 Verification of agent-based simulations

Verification approaches for agent-based simulations are still largely missing; a brief overview of existing work is given below. Due to the lack of formal techniques, we also include approaches based on testing and monitoring.

RatKit is an automated testing framework for agent-based simulations which has been presented recently [43]. It is based on the Repast simulation platform [63] and supports the formulation of user-specific test cases written in Java. RatKit allows for the repeated execution of test runs in a parametric or periodic way by means of code annotation. The execution of the simulation is observed by a *TestAgent* which stores time-stamped observation results and reports them to a central runner which itself stores the entire execution history together with test results. Test cases themselves are based on *observation definitions* and can be of *simple* or *aggregate* nature; the former refer to individual agent attributes, the latter to the result of aggregation functions such as $count$, max or min.

MASTER is a testing framework which is specifically designed to identify 'suspicious' simulation runs, i.e. simulation runs which deviate from the simulation's 'normal', expected behaviour [246]. It has been originally designed for MASON, a popular Java-based agent-based simulation platform [166] but has since also been extended for FLAME [138]. MASTER first observes a small set of runs in order to gain an understanding of the 'normal' behaviour of the simulation; this step is referred to as *capturing* and essentially represents initial calibration. The knowledge obtained during capturing is then used to find deviations in a much larger sample of runs. The level of deviation necessary in order to deem a run 'suspicious' needs to be configured by the user. Similar to the previous approaches, testing criteria are specified in the target programming language.

VOMAS, the *Virtual Overlay Multi-Agent System* approach, represents another, related approach to the monitoring of agent-based simulations [194]. The approach proposes an additional multiagent system, a system of testing agents, on top of the agent-based simulation. The testing agents are designed by subject matter experts and are used to monitor the simulation, to log events and to check the compliance with given constraints, i.e. invariants. The authors also propose a basic methodology describing how the

overlay system should be designed. Information on how the actual monitoring and invariant checking is being performed, however, is not given in the paper.

A different formal approach to the analysis and verification of agent-based simulations has been proposed by Izquierdo *et al.* [131]. The authors describe how simulations can be encoded into time-homogeneous Markov chains and analysed in terms of their transient and steady-state behaviour. In order to illustrate this approach, they use popular social simulation models such as Schelling's model of spatial segregation [218], Epstein's Sugarscape [89] or Axelrod's metanorms model [8]. Since the main focus of the paper is on the usefulness of Markov chain analysis for the understanding of complex simulation models, the authors do not provide any state space reduction techniques in order to circumvent the combinatorial explosion. However, they describe ways of analysing the behaviour without having to represent the transition matrices. More specifically, they make assumptions about the nature of the state space and derive insights by pure reasoning, e.g. whether the system will eventually reach an absorbing state and stay there forever.

An approach which combines agent-based simulation with numeric analysis has been presented by Wolf *et al.* [70]. The main motivation of their work is to determine whether an individual-based system exhibits certain macroscopic emergent behaviour. To that end, repeated simulation is paired with numeric analysis from the system dynamics domain in order to detect deviations or to approximate the steady-state behaviour of the simulation. An advantage of the approach is its capability to speed up the simulation process by *steering* it into the direction required by the respective analysis algorithm. Due to its global focus, the approach is restricted to macro-level analysis; nevertheless, the power lies in the fact that it allows for the analysis of properties which conventional testing is not able to deal with.

2.6 Summary

Despite the increasing adoption of agent-based modelling, formal approaches to their engineering — particularly for the purpose of verification — are still largely missing. In the more general area of multi-agent systems, a wide range of verification approaches which allow for correctness checking of various property types and on different observational levels have been presented; some of the seminal work has been described in Sections 2.5.1 and 2.5.2. Due to the conceptual proximity of multiagent systems and multiagent simulations, it is natural to consider the application of those approaches to simulations. For some aspects and models, this may well be possible and successful. However, the particular mix of

characteristics that simulations typically exhibit, for example large populations, heterogeneity and randomness, poses additional requirements which are not satisfied in their entirety by existing multiagent verification approaches.

In the modelling and simulation literature, on the other hand, the major focus of interest in quality assurance has traditionally been on external validation and empirical analysis. Verification (or internal validation) — if considered at all — has mostly been equated with conventional software testing [48]; some existing approaches in this area have been described in Section 2.5.3. Surveys have shown that, in practice, most modellers tend to rely on informal assessments and expert opinion rather than on any type of rigorous correctness checking [115]. As indicated in various places above, one of the possible reasons for the lack of dedicated verification techniques may be that agent-based modelling has primarily been employed by social scientists who have strong statistical skills but typically less expertise in software engineering, let alone in formal methods. In the computer science community, agent-based modelling has been largely neglected to date. The goal of this work is to address this problem and bring more rigour into the model engineering process by providing a dedicated verification technique. A helpful starting point in that respect is to start clarifying the different concepts used throughout the literature and to unify them under a common, conceptual framework. This is the purpose of the next chapter. We start with an informal description of typical characteristics and components of agent-based simulations, followed by a comprehensive formalisation using Object-Z. The resulting formal framework is used (i) to define the interface between a simulation and a monitor used as part of a runtime verification framework, and (ii) to define a notion of *events* which form the basis for verification properties.

Chapter 3

A Formal View on Agent-based Simulations

3.1 Introduction

The purpose of this work is to develop a runtime verification approach which allows for the verification of the temporal behaviour of arbitrary agent-based simulations in a timely manner. As described in Section 2.4.4, any runtime verification approach requires two essential components: the *system* to be verified and a *monitor*. In order to design an efficient monitor, its interface to the system that it is supposed to observe needs to be clearly defined. The goal of this chapter is to start clarifying the requirements and to formalise some of the concepts that are used in subsequent chapters on an abstract, yet rigorous level.

The central focus of interest in this chapter is on the system to be verified, i.e. the agent-based simulation. Given the highly interdisciplinary nature of the field, a plethora of different models, application domains, modelling techniques and tools can be found in the literature. In order to develop a general verification methodology, it is necessary to clarify these different meanings and streams of work and attempt to unify them under a common conceptual framework. The purpose of this chapter is to narrow the gap between the highly diverse, interdisciplinary and application-oriented world of agent-based modelling on one side and the area of formal methods which requires clear, unambiguous and mathematical descriptions of the types of problems it is dealing with on the other hand. This is done by

developing a formal and rigorous specification of the typical constituents of an agent-based simulation as well as its temporal dynamics. It is obvious that a general formalisation will never be able to cover all aspects of all models found in the real world. This is, however, neither necessary nor desirable here; rather than being *prescriptive* and developing a detailed software framework which can be practically used to construct models, we aim to be *descriptive*, distill common characteristics and describe them in an abstract, yet rigorous way with the overall purpose of *clarification* and the establishment of a *common language*.

In developing a formalisation for a field like agent-based modelling, a number of challenges need to be overcome. First, in an attempt to consider agent-based models in general without focussing on a particular domain, one faces — as indicated above — an overwhelming variety of philosophies, approaches, models, tools, frameworks, etc. The wider agent-based modelling community comprises researchers and practitioners from disciplines as widespread as economics, sociology, philosophy, medicine, computer science, physics and chemistry, to name but a few. In fact, it is hard to find a domain that agent-based modelling has not touched yet. Naturally, different schools of thought have different (and often conflicting) opinions about fundamental principles which makes it difficult to distil common characteristics. In identifying a common ground, finding a good balance between a high level of detail and a high level of abstraction is crucial. A model which is abstract enough to accommodate all possible theories and allows for arbitrary interpretations may be intellectually interesting but of little practical use – especially for formal analysis. A highly detailed model, on the other hand, may simply be too complex to be analysed and thus equally worthless.

The second component which needs clarification in a runtime verification context is the *monitor* whose purpose it is to assess the correctness of the simulation. We touch upon the structure of the monitor only very briefly in this chapter and describe it on a very high level of abstraction from an *external*, process-based perspective with a mere focus on its *interaction* with the simulation. A detailed description of the monitor's internal workings are given in Chapter 6.

The contributions of this chapter are as follows:

- We give informal high-level descriptions of the constituents of an agent-based simulation and translate them into formal specifications.
- We derive, from the abstract model, the notion of *simulation traces* as the central exchange data structure between the simulation and the monitor. Traces represent temporal sequences of global states which are themselves composed of the states of the agents at particular points in time.

- We describe formally how fragments of simulation traces correspond with *measurable events* which themselves form the basis for the formulation of *properties*.
- We show that simulation corresponds with *sampling from different spaces*, the choice of which depends on the questions that need to be answered. We illustrate that, depending on the chosen interpretation of the sample space, different types of events become detectable.

The idea of simulation traces developed in this chapter largely corresponds with the idea of *runs* described by Fagin *et al.* [91]. Runs represent an important component of *interpreted systems*, a formalism to model computationally grounded multiagent systems. It is important to note that, despite the conceptual similarity between some of the aspects of our framework and interpreted systems, we pursue a different goal. Since the focus of this work is on the *temporal verification of individual simulation traces* rather than on full model checking as, for example, done by MCMAS [163], a full formalisation of the underlying system is not necessary. Rather than presenting yet another formalism for *modelling* multiagent systems, the formal framework given in this chapter thus aims to serve a primarily illustrative purpose. In formalising the concepts and deriving a notion of traces, we aim to motivate and clarify the interface between a simulation and a monitor used for runtime verification.

The chapter is structured as follows. Section 3.2 gives an informal characterisation of agent-based models. Section 3.3 describes the structure of the proposed verification approach in an abstract, yet formal way. Section 3.4 provides a formalisation of the basic concepts in an agent-based model, followed by a description of the macro level and examples to show the representativeness of the described framework. A description of simulation traces as the central information exchange data structure between the simulation and the monitor is given in Section 3.5. Traces form the basis for the definition of *events* and, ultimately, *properties* as described in Section 3.6. The chapter concludes with a summary in Section 3.7.

3.2 Characteristics of agent-based simulations

Before describing the design decisions underlying the formal framework, let us briefly try to answer the question what an agent-based simulation actually is. Answering this question is a surprisingly difficult task. Despite the variety and ever-growing volume of agent-based models presented in literature, attempts at clearly characterising the field of agent-based modelling from a technical perspective and demarcating it from its neighbouring disciplines are still largely missing. It seems straightforward to view

agent-based simulations as a subtype of agent-based systems in general. Despite its logical plausibility, this statement does, in reality, not necessarily hold; some reasons have already been given in Sections 2.1 and 2.2. In fact, some agent-based simulation models (especially if they are highly data-driven) have much more in common with simple probabilistic processes, with Markov chains or with statistical models than with typical multiagent systems. Whether or not those simulation models deserve the label 'agent-based' is clearly debatable but beyond the scope of this work.

The following list is meant to serve as an informal collection of *typical* characteristics of agent-based simulations. It is neither restrictive in requiring *every* agent-based simulation model to exhibit *all* of the characteristics listed, nor by any means exhaustive. It results from several years of work in the agent-based modelling domain — both in an academic and in an industrial environment — and hopefully covers common properties which can be found in a wide range of models presented in the literature on a sufficiently high level of abstraction.

We start with a general description of typical constituents. An agent-based simulation (or agent-based model) is a software system which comprises a *population of agents* which are situated in an *environment*. The nature of the environment is not further specified and can range from a simple grid world to a comprehensive set of interconnected objects for the purpose of providing resources to the agent population. In many cases, the environment does not contain any logic and merely fulfills the purpose of 'hosting' the agents.

Agent-based models are generally constructed for the purpose of analysis. To this end, their dynamics are typically *highly regulated* and *centrally organised*. This represents a significant difference to the general area of multiagent systems, where agents are assumed to be entirely independent and essentially distributed. It is typical for an agent-based simulation to contain a *global clock* based upon which agents are updated in *discrete time steps* or *ticks*[1]. The actual interaction between agents can be of various kinds. One way is to let agents interact in a randomly pairwise fashion, an approach which is closely related with evolutionary game theory [237]. Alternatively, a modeller can choose to update only a single, randomly picked agent in a tick. What we often find in real models, however, is an update of the whole population within a tick, the internal order of which is determined by a *scheduling mechanism*.

Agent-based simulations are also typically *time-bounded*: after a finite number of time steps (which is, for example, either determined by the range of external input data available or by the time period

[1] A related, somewhat 'dual' approach is to define the progress of an agent-based simulation in terms of *discrete events* rather than *discrete time steps*. This offers the advantage of finer granularity with respect to time, i.e. continuous time values can be represented. For the purpose of the verification approach defined in this work, however, the distinction is irrelevant and shall thus not be further discussed here.

of interest), the simulation stops. And finally, in order to represent uncertainty as well as to introduce heterogeneity into the population, agent-based simulations typically exhibit a significant amount of *randomness*.

An interesting observation is that, in agent-based models, agents have a natural tendency to be rather *reactive* in nature. This is in contrast to the more deliberative nature of agents constructed for the purpose of actual problem solving, for example certain types of robots. The reason for that is, again, in the nature of agent-based models as tools for *scientific discovery*. In such a model, agents are simplifications of their counterparts in reality (whatever part of reality the model purports to represent) and therefore only 'sketched' in a mostly rather abstract way. Complex cognitive architectures such as BDI [101], Soar [150], or ACT-R [5] are rarely used in the area of agent-based simulation. The focus of analysis tends to be more on the overall behaviour of the system rather than on the internal workings of an individual. This is certainly not always the case but can be assumed in general. Another important implication of the analytic nature of agent-based modelling and their focus on emergent, macro-level phenomena is that they often contain a *significant number of agents* – often hundreds or thousands, in certain cases even millions.

Due to their exploratory nature, agent-based simulations are typically *not safety-critical*. This has important implications on the nature of verification & validation. In a safety-critical context, for example air traffic control or nuclear power plants, failures may realistically result in humans being injured or even dying. In this case, verification requirements are very strict and may — depending on the safety level — involve mathematical correctness proofs. In agent-based modelling, things are generally less strict. This is not to say that the dynamics of a simulation should not be subject to strict verification & validation — quite the contrary. After all, simulations are increasingly being used as a basis for policy-making and should therefore also be sufficiently correct and valid. However, instead of *strictly proving* that the system does not exhibit certain behaviours (which would still be desirable but, given the complexity, is largely infeasible), approximate solutions which do not prove the *absence* of errors but increase confidence by showing that the *presence* of errors is sufficiently unlikely are often perfectly acceptable.

3.3 Design and analysis of agent-based simulations

The purpose of this work is to develop a *statistical runtime verification framework* for agent-based simulations. In runtime verification, the program to be verified is augmented with a *monitoring process* which obtains runtime information from the program and checks its correctness on-the-fly; as soon as a violation occurs, the execution is stopped. In our case, the program to be verified is an agent-based simulation and, since the verification of individual traces is based on temporal logic model checking, our monitoring process is essentially a model checker. In essence, our runtime verification framework can be seen as a system comprising *two interacting processes*. In order to pose as few restrictions upon the types of simulations to be verifiable as possible, we assume that both processes are largely decoupled and independent. The simulation does not need to know anything about the internal workings of the monitor in order to function correctly. Likewise, for the purpose of this work, the monitor does not need to know *how* the simulation framework works internally, how agents see the world and how their internal decision making process has been designed. All they need to do is to agree on a protocol which they follow in order for each process to obtain the information necessary for their correct functioning.

The purpose of this chapter is to formalise these up to now informally described ideas. Due to the structure of the framework as a system of interacting processes, it is useful to employ a process-centric formalism for their specification. Process algebras or process calculi, most notably Hoare's *Communicating Sequential Processes (CSP)* [126] and Milner's *Calculus of Communicating Systems (CCS)* [189] have been designed to model concurrent and distributed systems in a formal way and prove properties about their dynamics. CSP will be used in this chapter, yet merely for illustrative purposes. We use its powerful refinement checking capabilities only very briefly to show equivalence between processes. Similar to other approaches, CSP models are also subject to combinatorial explosion and therefore not applicable to the verification of agent-based simulations. Nevertheless, as shown below, CSP can still be employed very usefully to *describe* a complex model in a succinct an elegant way. An overview of the CSP notation is given in Section 2.3.2.

Let us start by describing the problem on a very abstract level. Based on the characterisation of agent-based simulations given above, we can describe such a simulation — from the point of an external observer — as a process which performs regular $tick$ events until a maximum number of time steps has elapsed. We assume that the observer is not able to examine the internal state of the simulation process itself; instead, the simulation process *reports* its internal state to the outside world exactly once every tick. In order to formalise this idea, we first need to introduce a number of data structures. Let

MAX_TICK denote the maximum time step of the simulation. Further let $GState$ denote the set of possible global states that the simulation process can be in; this set can be understood as the Cartesian product of all the individual agents' state spaces and is described in more detail in Section 3.4.1. We also assume that there is a certain initial state $init \in GState$. A generic simulation process DTS (for **D**iscrete **T**ime **S**imulator) can then be described as the following parametrised CSP process:

$$
\begin{array}{l}
DTS = \\
\quad \text{let} \\
\qquad State(t, s) = \\
\qquad\quad \text{if } t == MAX\_TICK \text{ then } tick!t?\_ \text{ then } STOP \\
\qquad\quad \text{else } tick!t?s' : GState \rightarrow State(t+1, s') \\
\quad \text{within} \\
\qquad State(1, init)
\end{array}
$$

The DTS process keeps two internal state variables, t and s. $t \in \mathbb{N}$ is a natural number which represents an internal time counter, $s \in GState$ represents the internal state of the process. Starting in the initial state, the process performs $tick$ events continuously until the maximum number of time steps has been reached. After each tick event, the internal time counter is increased by 1. The $tick$ event has two parameters: the first parameter represents the time step and allows the process to synchronise with a global clock; the second parameter allows the process to communicate its internal state to the outside world.

We can also represent an abstract version of the monitoring and verification process in a similar way. We assume that it engages with the DTS process such that, every tick, it *receives* the current state of the simulation, verifies a given property upon it and, as soon as the property is either satisfied or refuted, reports the result to the environment and stops. Similar to DTS above, the monitor can be represented by a process MON which is able to engage in two events: $tick$ (together with DTS) and $done$. In order for the process to return a useful result, we assume the existence of a set $Result = \{true, false\}$ which describes the possible outcomes of a verification check, $true$ and $false$. The process can then be described as follows:

$$
\begin{array}{l}
MON = \\
\quad \text{let} \\
\qquad State_1() = \\
\qquad\quad (tick?t?s : GState \rightarrow State_1()) \sqcap State_2() \\
\qquad State_2() = \\
\qquad\quad \bigsqcap r : Result \bullet done!r \rightarrow STOP \\
\quad \text{within} \\
\qquad State_1()
\end{array}
$$

In this abstract description, the monitor does not keep any internal data. It merely engages in $tick$ events, as part of which it receives an instance of $GState$. After that, it performs a decision as to whether evaluation needs to continue or not; details of this decision process are omitted here and simply represented as an *internal choice* (using symbol '$\sqcap$'). Once evaluation has finished, the process makes another internal choice as to whether the property being evaluated is satisfied or refuted.

The overall runtime verification framework can then be abstractly represented as a *composite process* RV which is defined as the *parallel composition* of DTS and MON, synchronising on the $tick$ event:

$$RV = DTS \underset{tick}{\parallel} MON$$

The descriptions so far have been rather abstract and deliberately neglected the internal structure of the respective processes. The purpose of this chapter is to partially address this problem and give a detailed, formal account for the representativeness of process DTS as an abstract model of agent-based simulations; as indicated above, the internal workings of process MON are described in Chapters 6 and 8.

In order to describe the internals of DTS, we follow a two-step approach which starts with an object-oriented description of the individual components of an agent-based simulation — the agents and the environment — using *Object-Z* [226].

The choice of Object-Z over conventional Z was motivated by several factors. First, due to its object-oriented capabilities, Object-Z allows for the elegant formalisation of encapsulated entities by means of *class schemas*; this aspect is particularly helpful for the design of agents, their attributes and operations. Second, Object-Z allows a modeller to bridge the gap between different levels of abstraction by providing individual-level, operational descriptions through class schemas and a more holistic view on the component's interactions through its integration with *Communicating Sequential Processes (CSP)* [125], a well-known formal notation for concurrent and distributed systems (see Section 2.3.2). This aspect is particularly helpful for the description of a population as a parallel composition of individual agents. Object-Z thus serves well as a mediator between an informal, high-level and entity-based description and one which is closer to the underlying and more abstract transition system semantics. As such, it should be fairly comprehensible for anyone with an object-oriented background without giving up its formal rigour. Furthermore, just like conventional Z, Object-Z is *state-based*, i.e. it allows one to focus on the temporal progression of the state of a system. As such, it is well suited to describe the

state transition semantics of systems which are subject to state-based verification, for example temporal logic model checking as described in this work.

Object-Z is an effective formalism for describing systems and their inherent state. For the description of concurrent and distributed systems — systems which comprise multiple (possibly complex) interacting components — however, other formalisms such as process algebras are better suited. Based on the integrative semantics of Object-Z and CSP described in Section 2.3.2, we thus translate the individual component descriptions into CSP processes and compose them together in order to represent the full agent-based simulation (as introduced above). The resulting process serves as a *succinct representation* of the entire simulation and describes its dynamics in a compact and more elegant way than an Object-Z description of the system would.

The formalisation presented in the following section was largely influenced by the work of Luck and d'Inverno [165, 79] and Miller and McBurney [188]. However, since our overall focus is on the purely temporal behaviour of large populations of agents, we have chosen to follow a slightly more abstract approach and omit some of the details of the agent internals. For example, instead of assuming a certain cognitive architecture and representing an agent's mental state by means of beliefs, desires, intentions, etc., we just view it as a collection of attributes — regardless what their conceptual meaning might be.

3.4 A formal framework for agent-based simulations

The next two sections are devoted to the formal descriptions of agent-based simulations in an abstract, yet mathematically rigorous way. As mentioned above, the purpose of the framework is *not* to be practically usable as a means to design or even implement simulations; it is of purely descriptive purpose with the overall goal of clarifying concepts and terminology. An overview of the Z notation used here can be found in Appendix A.

Before a the basic components are described formally further below, some general definitions need to be given. In order to describe the attributes, percepts, etc. that an agent as well as the environment may possess in an abstract way, we assume the existence of the following two sets: $Name$ (set of possible attribute names), and $Value$ (set of possible attribute values).

We further define an *attribute* as a simple key-value pair containing a name and an attribute value, both of which are drawn from the sets defined above:

$$Att == (Name \times Value) \tag{3.1}$$

3.4.1 Environment

An important component within an agent-based simulation is the *environment* [239]. We view the environment as an individual component that describes the world in which an agent is situated and which it is able to perceive. The concrete nature of the environment differs from model to model and depends both on the problem domain and the characteristics of the respective model. Nevertheless, a typical and rather general task of the environment is to store and provide resources for which the agents compete. In addition to just acting as a resource and service provider, however, the environment may as well exhibit its own individual behaviour. We thus view the environment as an active rather than a purely passive component.

Apart from providing resources, the environment can also be viewed as a central hub of *interaction*. Interaction between constituents represents a central feature of complex systems which causes global phenomena to emerge from simple local behaviour. The interaction itself can be of arbitrarily complex nature and range from simple perception to extensive communication based on speech act theory [220] or the exchange of complex data structures, e.g. ontologies. It is thus not surprising that agent communication represents an active research area in its own right. In an effort to formalise aspects of agent-based simulations, finding a good balance between richness of detail and generality is one of the most frequently recurring challenges, also in the case of communication. In order to avoid having to deal with details of particular communication mechanisms, we decided to replace direct communication between agents with *perception* and *indirect action* according to which agents are only allowed to affect the environment and not their neighbours. Even though, in a concrete implementation, communication will most likely be realised in a direct manner, we have chosen to follow the indirect path for this work due to reasons of simplicity. Direct communication would have required additional arguments being passed on the communication channels as well as potential reception and acknowledgment mechanisms. These aspects would have complicated the specification significantly without adding useful information.

In order to be general enough, we abstract away from any particular details and view the environment as a simple state-based entity whose state is defined as a function from the set of attribute names to the set of attribute values:

$$EState == Name \rightarrow Value \tag{3.2}$$

In addition to its own state, the environment also keeps a record of the *global state*, i.e. the state of the agent population. This is necessary since, as described above, we assume that all communication between agents happens via the environment. Formally, a global state — which we, for reasons given further below, refer to as a *group state* — can be described as a function from the set of agent identifiers Ag to the set of agent states $AState$:

$$GState == Ag \rightarrow AState \tag{3.3}$$

Finally, the environment is aware of the number of agents in the system, as denoted by the attribute $numAgents$. This is merely a technical necessity which allows the environment to perform its own state update if and only if all agents have finished their perception stage, as described further below. The overall state of the environment can thus be described as follows:

Environment

$attributes : EState$
$popState : GState$
$numAgents : \mathbb{N}$

The environment further needs to be able to be perceivable by the agents and to update its own state based on its internal logic. The former is accomplished by an operation $Perceive$ which allows the environment to share its state with the agents. The signature of the operation is equivalent to that of an equally named function of the agent described further below. This allows both components to synchronise and to exchange information. In this case, the environment *sends* data to the agent (denoted by the out-parameter $\_es$):

Environment

Perceive

$\_id? : Ag$
$\_es! : EState$

$\#popState < numAgents$
$\_es! = attributes$

The first line in the constraint part of the schema describes that the $Perceive$ operation is enabled as long as the size of the sequence $popState$ is strictly less than the overall number of agents in the system. This is, again, a merely technical trick. It enables the environment to allow any external component to engage with its $Perceive$ action if and only if not all agents have yet performed their perception step.

The update of the environment's internal state happens in the $Update$ operation which is shown below. The operation accepts three parameters: an agent id ($\_id$), an agent state ($\_as$) and an environment state ($\_es$). This signature is equivalent to the one in the equally-named operation of the $Agent$ class described further below. Equal signatures allow operations to *synchronise* and to exchange information. In this case, the environment *receives* data from the agent (denoted by in-parameters $\_as$ and $\_es$). The environment stores the received information about the agent's state in its own $popState$ attribute. The changes received from the agent are applied on top of the environment's current state by means of a not further specified function $update$ which causes it to transition into a new state:

Environment

Update

$\Delta(attributes, popState)$
$\_id? : Ag$
$\_as? : AState$
$\_es? : EState$
$update : (EState \times GState) \rightarrow EState$

$\#popState < numAgents$
$popState' = popState \oplus \{\_id? \mapsto \_as?\}$
$attributes' = update(\_es?, popState)$

Finally, since it is one of the responsibilities of the environment to represent the state of the population, there needs to be a way to make this information accessible to the outside world. To this end, we introduce a $Tick$ operation which denotes the completion of a time step. It is only enabled if all agents

have been updated (i.e. if the length of the sequence $popState$ is equal to the number of agents in the population). The environment can use this operation to pass the population state to an external observer:

Environment

...

Tick

$\Delta(popState)$
$tick? : \mathbb{N}$
$\_ps! : GState$

$\#popState = numAgents$
$popState' = \langle\rangle$
$\_ps! = popState$

The usefulness of $Tick$ will become clearer when the Object-Z specifications are translated into CSP processes further below.

This concludes the description of the environment. The next section discusses the most fundamental entity of an agent-based simulation model — the agent.

3.4.2 Agent

The central component of any agent-based model is the agent. As described in Section 2.1, there is no common agreement about what constitutes an agent and what distinguishes it from adjacent concepts such as, for example, software objects. This uncertainty equally holds for the area of agent-based simulation models, albeit to a lesser extent. As already mentioned further above, in a simulation, agents are typically more constrained and less autonomous than their non-simulation counterparts. Often enough, the behaviour of an agent in a simulation is driven by historical data and therefore largely predetermined. In the presence of historical data, it is not uncommon for the entire decision making process to be fully 'baked' into its transition probabilities; in this case, the agent regresses to a simple probabilistic process. The focus of interest in those entirely probabilistic models can be generally seen as being largely on the emergence of higher-level phenomena from the interaction of a group of heterogeneous individual entities rather than on the problem solving capabilities of a single agent or a group of agents in a potentially unknown environment. Whereas academia tends to focus more on behavioural types of models — 'generators' in Heath's taxonomy (see Section 2.2.2) — their data-driven counterparts ('mediators' or 'predictors') are particularly prevalent in the commercial world.

In order to abstract away the actual mode of implementation of an agent's behaviour, we take a simple state-based view on the nature of agents here and model their internal state as a function from the set of attribute names to the set of attribute values (similar to the state of the environment described above):

$$AState == Name \rightarrow Value \tag{3.4}$$

Those attributes could stand for beliefs, desires, goals, plans, logical predicates, numeric values, etc. At each point in time, an agent is in a particular state which is determined by the valuation of its attributes. Agents perform *actions* in order to change their internal state or to affect the environment. We model an action as a partial function from an agent state to its successor state. The way in which the transformation takes place depends on the problem and is thus omitted here:

$$Action == AState \nrightarrow AState \tag{3.5}$$

An agent is modelled as an entity with both fixed and variable attributes, the latter of which can be seen as the agent's state. Fixed attributes are the agent's unique id and a set of *capabilities*, i.e. actions that the agent is able to perform. The variable comprises a sequence of attributes as well as a set of environment percepts of type $EState$[2] which is assumed to be empty initially:

Agent

$agentId : Ag$
$capabilites : \mathbb{F}\ Action$

$attributes : AState$
$envPercepts : EState$

INIT

$envPercepts = \varnothing$

The dynamics of an agent are governed by a simple update cycle. Even though the nature of the update cycle may vary from simulation to simulation, we assume that it will follow the canonical *sense-select-act* schema, according to which the agent performs four basic tasks [212]:

1. sensing / perceiving the environment

[2] $EState == Name \rightarrow Value$.

2. updating the internal state
3. action selection
4. acting
   (a) internal acting
   (b) external acting

All steps are described in more detail and formalised in the following subsections.

Sensing / perceiving the environment: At the beginning of each tick (i.e. logical time step), an agent perceives the world in which it is situated. Agents are often limited in terms of their perception capabilities, as a consequence of which their knowledge about the world may only be partial. The actual type of perception may also vary from system to system and range from perception of the environment or an agent's neighbourhood to extensive communication based on speech act theory [220] or the exchange of complex data structures such as ontologies. For simplicity, we view interaction between two components (e.g. two agents or an agent and the environment) as a *handshake* operation, i.e. as an operation which all actors need to participate in [12]. In this case, the participating actors are the agents and the environment. Perception on the agent side is modelled as an operation $Perceive$ which is able to receive information from the environment (through the input parameter '$\_es$') and to store it in the agent's perception store $envPercepts$:

Agent

Perceive

$\Delta(envPercepts)$
$\_id! : Ag$
$\_es? : EState$

$envPercepts' = \_es?$

type of behaviour allows an agent to *observe* its environment over a longer period of time before acting. The state update can thus be viewed as a process of moving items to the long-term memory.

Formally, the state update is modelled as an operation $StateUpdate$. It comprises a (not further specified) function $mergePerc$ which merges the environment percepts with the agent's current state:

Agent

StateUpdate

$$\Delta(attributes)$$
$$mergePerc : (AState \times EState) \rightarrow AState$$
$$attributes' = mergePerc(attributes, envPercepts)$$

Action selection: The next step comprises the selection of a set of actions for execution. The actual selection is, of course, dependent upon the logic of the respective model and thus only described in an abstract way here. In the case of a purely rational agent, for example, the choice of action is typically based on a utility evaluation. Planning agents will select actions from a plan library instead. Although indispensable for the *implementation* of an agent-based simulation, we abstract away from any details regarding the decision making process here and describe action selection in general terms as a mapping from the agent's current state, its percepts, and its capabilities to a finite set of possible actions to choose from:

Agent

ActionSelection

$$willdo! : \mathbb{F}\ Action$$
$$selectActions : AState \times EState \times \mathbb{F}\ Action \rightarrow \mathbb{F}\ Action$$
$$willdo! = selectActions(attributes, envPercepts, capabilities)$$

In this operation schema, action selection is described as a *nondeterministic choice*. Nondeterminism can arise — knowingly or inadvertently — from abstraction. In this case, nondeterminism is used to abstract away from the underlying decision making process and describe the concept of choice and uncertainty. In an agent-based model, uncertain decision making is often implemented as a probabilistic choice which is further described in Section 3.6. The important point to be conveyed here is that, starting from any state, the action process of an agent may cause it to transition into a range of possible successor states — regardless what the underlying decision making process may look like.

Acting: The agent can use the chosen actions in order to (i) update its own state again (*internal action*) and (ii) affect the environment (*external action*). Internal action can be understood as applying the set of chosen actions on the current state in order to perform an update. In a concrete model, the set of actions may need to be composed in a particular way; this is abstracted away here and represented by a not further specified function $actInt$:

Agent

ActInternal

$\Delta(attributes)$
$\_id! : Ag$
$\_as! : AState$
$willdo? : \mathbb{F}\ Action$
$actInt : Action \times AState \rightarrow AState$

$attributes' \in \{ws : willdo? \bullet actInt(ws, attributes)\}$
$\_id! = agentId$
$\_as! = attributes'$

The same logic can be applied to the case of an agent's external acting. Operation $ActExternal$ describes an agent's acting upon the environment by creating a set of environment attributes and passing them over to the environment (by means of the out parameter $\_es$). Similar to internal action, the actual decision about *what* information to pass on is left open and described in an abstract way by a (not further specified) function $actExt$:

Agent

...

ActExternal

$willdo? : \mathbb{F}\ Action$
$\_es! : EState$
$actExt : Action \times AState \rightarrow EState$

$\_es! \in \{ws : willdo? \bullet actExt(ws, attributes)\}$

Logically, internal and external action can be seen as a single action operation. In order to reflect this idea, the two separate operation schemas $ActInternal$ and $ActExternal$ can be combined into a single operation schema Act using schema calculus as follows:

$$Act \mathrel{\widehat=} ActInternal \land ActExternal \tag{3.6}$$

In addition to the update of the agent's internal state, both functions also produce output: the agent id, the agent's current state and the calculated change to the environment. This allows the agent (i) to propagate its state to the environment in order for the environment to keep track of any changes, and (ii) to influence the environment as necessary. Both aspects are described below.

Specifying operations separately (like $ActInternal$, $ActExternal$ and $ActionSelection$ above) emphasises their logical independence and increases readability. However, in order to highlight their indistinguishability to an external observer, it is useful to ultimately combine them into a single conceptual operation (as we did for operation Act above). Due to an agent's autonomy, internal actions are, in fact, indistinguishable to an outside observer. From an external point of view, an agent performs just two steps in a tick: *perception* (input) and *update* (output). This can be reflected in the formal description by further combining action selection, internal and external action (which has already been aggregated into a combined schema Act above) into an overall $Update$ operation, using the parallel composition operator '$\|$', as follows:

$$Update \mathrel{\widehat=} ActionSelection \parallel Act \tag{3.7}$$

As described above, $ActionSelection$ produces a set of actions ($willdo$) which is consumed by the combined operation Act. This intermediate data structure is entirely internal and thus not visible to an outside observer. As a consequence, the operation schema $Update$ resulting from the combination of $ActionSelection$ and Act has the following signature:

Update

$\Delta(attributes)$
$\_id : \mathbb{N}$
$\_as! : AState$
$\_es! : EState$

...

Because of common operation arguments, $Perceive$ and $Update$ are now able to communicate values to the equally named operations of the environment class described in the following section. This allows the agent and the environment to synchronise in order to exchange information and, in doing so, model perception and external action.

Autonomy

Before the Object-Z classes described above are translated into a more concise, process-algebraic description in the following section, we briefly revisit the notion of *autonomy* and show how it is reflected in the formal specification given above. Autonomy is widely recognised as one of the distinctive characteristics of agency. Since it is not uncommon for an agent-based simulation to consist of agents whose behaviour is purely reactive or even entirely predetermined (as described in Section 3.3), the notion of autonomy has a different flavour in a simulation context and is often less strict. Among the various different definitions of autonomy in literature, frequently recurring aspects comprise an agent's ability to act on its own, to have control over its internal state and actions and to pursue its own agenda based on individual goals and plans. In the following paragraphs, we describe how these rather informal definitions of agency are reflected in the formal specification given above.

Aspect 1: An autonomous agent acts on its own: The possibility to act on its own is a characteristic feature of agent autonomy. As opposed to other Z-based formalisms [172], Object-Z does not provide a direct means to specify systems which have their own thread of control. However, the agent class is modelled in such a way that the choice of action is not directly influenceable from outside but based upon an agent's percepts, its current state and an internal choice function which has been deliberately kept unspecified. In this way, agents are given their own thread of control which is a technical analogy to making them autonomous entities which act on their own. Furthermore, choice of action has been modelled as an internal and nondeterministic choice. Especially on the process level, where the nondeterminism arising from the agent's internal choice of action prevents an external process from influencing the temporal progression of an agent directly, the autonomous nature of the decision making process becomes apparent.

Aspect 2: An autonomous agent has control over its internal state: Above, we described an agent's decision making process as an action selection process which is influenced by its own attributes and percepts. Against this background, the agent's level of autonomy determines to what extent external influences cause it to deviate from the state transition that it would have taken without the external influence. In the case of a purely reactive agent, each state transition is solely determined by the agent's percepts. On the other hand, a purely autonomous agent would make its decisions solely based on its own internal state, without being influenced by its environment or its neighbours. We do not specify the decision making autonomy explicitly in our formal specification and hide it instead within the $ActInternal$ function.

Aspect 3: An autonomous agent pursues its own agenda based on individual goals and plans: This aspect is less obvious since it includes higher-level notions like goals and plans which are not represented explicitly in the formal framework. However, we can assume that all notions such as beliefs, goals, desires, intentions, are — however abstract they might be — eventually encoded as a set of variables. In the formalisation given above, this set of variables is represented by the set of attributes within the agent class. Since the action selection depends on these attributes, we can infer that the decision making process may also involve goals and plans which are implicitly stored in the state and thus allow the agent to pursue its own individual goals and plans.

This concludes the formalisation of the individual components. The verification approach in this work focusses on the temporal evolution of the system; from the monitor's perspective, the internal decision making process of an agent is thus entirely opaque. Object-Z as a state-based formalism is not well suited to describe the interaction of processes on an abstract level; in order to achieve that and get closer to an 'external' view on the interaction between agents and the environment, the Object-Z specifications are translated into CSP-based process descriptions in the next section.

3.4.3 Process algebra description

Starting with informal descriptions, the previous section provided formal Object-Z specifications of the basic components of an agent-based simulation — the agent and the environment — with the overall purpose of defining their state-based, temporal dynamics. In order to shift our focus to the interaction between agents and the environment, we now translate the Object-Z specifications into parametrised CSP processes using the approach described in Section 2.3.2.

Before we describe the translation, it is important to briefly introduce the distinction between an agent's *cognitive state* and its *control state*. A cognitive state can be understood as the content of the agent's memory, i.e. its $attributes$ (which could describe beliefs, goals, etc.). A control state, on the other hand, refers to the state of the associated *process* and can be seen the agent's current position within its own behavioural protocol. As described above, an agent typically performs several operations in a certain order during its sense-select-act cycle. Each position within this cycle can be seen as a particular control state which determines which operation is to be performed next. The agent specification given above is still fairly lenient in terms of the order of operations; it is, for example, possible for an agent to perform an indefinite number of subsequent $update$ events. A solution to that is provided by a *scheduler*

described further below. The distinction between cognitive and control states is important because their cardinalities are rarely the same. As we see further below, due to an agent's internal dynamics, the number of control states is typically much higher than the number of cognitive states. From the perspective of the monitor which is concerned with the agent's *behavioural evolution*, however, the cognitive states are the ones which are relevant. This is an important requirement for designing the interaction between simulation and monitor.

The first component to be translated is the $Agent$ class. From a verification point of view, we are interested in the temporal evolution of an agent's cognitive state, i.e. its set of attributes. We can thus ignore helper structures such as *capabilities* or *willdo* and model the agent class as a parametrised process $Agent(i, s)$ where $i : \mathbb{N}$ denotes the agent's id and $s : AState$ its cognitive state. Operation $Perceive$ has two arguments: one out-argument $\_id$ and one in-argument $\_es$. As described in Section 2.3.2, there is a direct correspondence between Object-Z operation schemas and CSP events. $Perceive$ can thus be modelled as an event $perceive : \mathbb{N}.EState$[3] with two channel parameters: the agent id and an instance of $EState$ (the environment state). Likewise, operation $Update$ corresponds to event $update : Ag.AState.EState$. From an external point of view, the $perceive$ event does not seem to cause any change in state to the agent; it only changes its control state which, however, is invisible to an external observer. The perception stage can thus be modelled as follows[4]:

$$Agent(i, s) = perceive.i?\_ \rightarrow Agent(i, s)$$

The situation is different for event $update$. Depending on what an agent perceives, it may decide to perform different actions; to an external observer, the choice of action is thus not influenceable. This can be represented formally by means of an *internal* or *nondeterministic choice* using CSP's '$\sqcap$' operator. Instead of representing the choice of action and the subsequent state update as independent processes, however, we collapse them into a single nondeterministic choice about the successor state:

$$Agent(i, s) = \sqcap s' : AState \bullet update.i!s'?\_ \rightarrow Agent(i, s')$$

The update event causes the agent to transition into a new cognitive state, as denoted by the change from s to s' in the process description.

[3] According to conventions, we start the name of operation schemas with an uppercase and the name of CSP events with a lowercase letter.

[4] '_' is used to denote an irrelevant channel parameter.

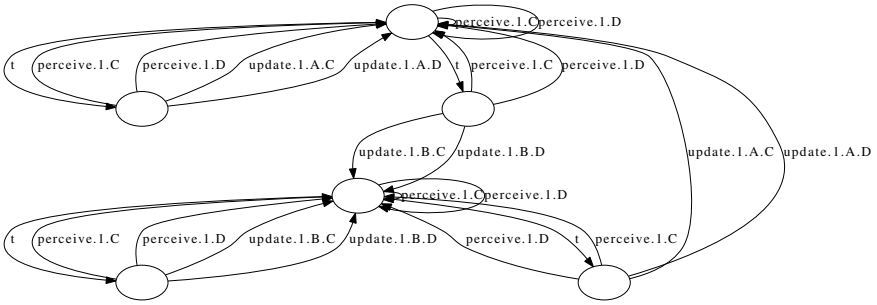


FIGURE 3.1: State space of the $Agent(1)$ process for $AState = \{A, B\}$ and $EState = \{C, D\}$

The Object-Z specification of the $Agent$ class did not make any assumption about the order of $Perceive$ and $Update$. From an external point of view, both actions are always enabled which can be represented formally as an *external choice* using CSP's '$\Box$' operator. Assuming that 'A' represents the initial state of the agent, the overall CSP process of the $Agent$ class can thus be given as follows:

$$
\begin{array}{l}
Agent(i) = \\
\quad \textbf{let} \\
\qquad State(i, s) = \\
\qquad\quad perceive.i?\_ \rightarrow State(i, s) \\
\qquad\quad \Box \\
\qquad\quad \bigsqcap s' : EState \bullet update.i!s'?\_ \rightarrow State(i, s') \\
\quad \textbf{within} \\
\qquad State(i, A)
\end{array}
$$

Class $Environment$ can be translated accordingly. We model the internal update of the environment as an internal choice over the set of possible states. Once all agents have performed their update operations, operation $Tick$ is enabled which is reflected in event $tick$:

$$
\begin{array}{l}
Env(s) = \\
\quad \textbf{let} \\
\qquad State(s, ps) = \\
\qquad\quad \textbf{if } \#ps < NUM\_AGENTS \textbf{ then} \\
\qquad\qquad perceive?\_.s \rightarrow State(s, ps) \\
\qquad\qquad \Box \\
\qquad\qquad \bigsqcap s' : EState \bullet update?\_?as!s' \rightarrow State(s', \langle as \rangle \frown ps,) \\
\qquad\quad \textbf{else} \\
\qquad\qquad tick?\_!ps \rightarrow State(s, \langle\,\rangle) \\
\quad \textbf{within} \\
\qquad State(s, \langle\,\rangle)
\end{array}
$$

Let us now briefly analyse the nature of the underlying transition system. The state space of each CSP process can be modelled as a directed graph in which the nodes denote states and the edges denote events. For illustration, we assume that the agent can only be in one of two cognitive states 'A' and 'B', i.e. $AState = \{A, B\}$, and that the environment can be in one of two cognitive states 'C' and 'D', i.e. $EState = \{C, D\}$. The graph that corresponds with the state space of process $Agent(1, s)$ where $s \in AState$ is shown in Figure 3.1. The nodes in the graph depict the agent's control states and the edges depict its operations described above. Internal, nondeterministic transitions which arise from internal choices are not observable by external processes and denoted by 't'. It becomes apparent that, due to its internal dynamics, the process has significantly more control states than cognitive states (6 as opposed to 2). From an observer's point of view, however, those internal dynamics are not visible; due to an agent's autonomy, the environment, for example, cannot 'see' intermediate states which occur during its reasoning process, for instance between an agent's perception and update stage. What it can see, instead, is a process which appears as exhibiting two observable states only: one, in which the agent announces its being in state A and one in which it announces its being in state B, both announcements of which are made via event $update$. Due to the agent's autonomy, the choice of order of those events cannot be influenced by an observer and should thus appear as being entirely 'arbitrary'.

The problem is that, in the current version of the processes, all events *are* still visible to an external observer. In order to change the $Agent$ process so that its external interface only shows what is relevant to an observer, we need to make $perceive$ unobservable to any other process by *hiding* it. Hiding is an important means of encapsulating functionality by making particular actions invisible to a process' environment [211]. It is done by using CSP's hiding operator '$\setminus$':

$$Agent'(i, s) = Agent(i, s) \setminus \{perceive\}$$

For any external observer, $Agent'$ now looks like a process which only performs $update.1.A$ and $update.1.B$ in arbitrary order — this is precisely what we want. In order to prove this formally, we define a second testing processes which exhibits the stipulated behaviour and show that it is *trace equivalent* to $Agent'$ [211]. In general, if two processes P and Q are trace equivalent, (denoted $P \equiv_{\mathrm{T}} Q$), then every trace (i.e. sequence of states) of P is also a trace of Q and vice versa:

$$P \equiv_{\mathrm{T}} Q \Leftrightarrow traces(P) = traces(Q) \tag{3.8}$$

.

Trace equivalence implies that both P and Q appear as equal from external observer's perspective. The testing process for process $Agent'$ can be described as follows:

$$AgentBehaviour(i) = \sqcap s' : AState \bullet update.i.s?\_ \rightarrow AgentBehaviour(i)$$

We do not give a formal proof here but, using a refinement checker like FDR3 [210], it can be shown that, in fact, $Agent'(i, s) \equiv_{\mathrm{T}} AgentBehaviour(i)$.

Due to the agent's autonomy, transitions between control states are no longer noticeable. When observing the agent's behaviour, however, any external process will now see the agent regularly 'publishing' its internal cognitive state via the $update$ event. We refer to such a sequence of agent states as an *agent trace*. We denote with Tr_a the set of all those agent traces.

$$Tr_a == \text{seq } AState \tag{3.9}$$

The environment collects the agent states which it receives from the individual agents via the $update$ event and uses them to assemble and emit a *group state* once every tick via event $tick$. This sequence of group states being emitted regularly by the environment represents an important data structure that the monitor uses to verify the properties of the entire population or of arbitrary groups of agents. In the remainder of this work, we refer to this sequence of states as a *simulation trace*; it is described in more detail in Section 3.5. A simulation trace can be decomposed into individual agent traces which can be used to verify properties of individual agents.

Despite both the environment and the agents engaging in common events, the actual interaction between the processes has not been modelled yet. This is done in the next section.

3.4.4 System level

This section extends the focus of analysis to the system level which is characterised by the behaviour of the agent population in collaboration with the environment. As opposed to previous attempts to the formalisation of multiagent systems [165, 79, 188], we employ a process-centric view. Instead of modelling macro-level concepts such as the population as a separate class (as we did with agents and the environment), we describe them as *compositions of individual components*. We start the description with

the population in the next section, followed by a description of the *world* representing a combination of the population and the environment. In order to reflect different orders of execution, the behaviour of the entire simulation is then described as a combination of the world and an additional scheduling mechanism. In order to illustrate the scheduler's internal logic, Object-Z specifications are also given.

3.4.4.1 Population

The central concept on the macro level of an agent-based simulation is the *population* which represents a set of concurrently acting agents. Since, as described above, we assume that agents only communicate via the environment, a population can be modelled as a simple parallel composition of n agent processes where all individual actions are entirely independent and thus fully interleaved:

$$Pop = Agent(1) \;|||\; Agent(2) \;|||\; ... \;|||\; Agent(n) \tag{3.10}$$

Using the set Ag of agent ids, the same process can be expressed more succinctly by using a distributed interleaving operator as follows:

$$Pop = \Big|\Big|\Big| \, i : Ag \bullet Agent(i) \tag{3.11}$$

3.4.4.2 World

The dynamics of the agent population represented by process Pop are rather simple: agents act concurrently, in arbitrary order and without ever receiving any information from the environment during the perception stage. This problem is solved by allowing the agent population to synchronise with the environment process Env in order for it to be perceived. In this case, synchronisation is used to describe interaction: every time an agent performs its $Perceive$ action, the environment participates in that action in order to exchange information. Formally, this joint activity can be modelled as a parallel composition between the Pop and Env processes by synchronising on the common $perceive$ event. We refer to the resulting process as the *World* process:

$$World = Pop \underset{perceive,update}{||} Env \tag{3.12}$$

3.4.4.3 Scheduler

Although the $World$ process models the interaction between the agents and the environment (and thus reflects the most central behaviour of an agent-based model), it still poses two problems: first, it allows for arbitrary combinations of individual agent actions, i.e. perception and update. Agents still have a high level of freedom with respect to the interleavings of their individual actions which allows sequences of agent operations to be noncoherent, i.e. interrupted by operations of other agents, or in the wrong order. It is, for example, possible to perform all $update$ events prior to performing $perceive$, as shown by the following example trace (for simplicity, only the first channel parameter, the agent id, is shown):

$$update.1 \rightarrow update.2 \rightarrow ... \rightarrow perceive.2 \rightarrow perceive.1 \rightarrow ... \tag{3.13}$$

If perception is required to happen prior to update, then this trace is clearly undesirable. It is also problematic since it allows for unfair behaviours, i.e. the repeated update of a single agent. Furthermore, since there is no upper bound on the number of updates that an agent performs, it will never terminate; the traces of the $World$ process are thus infinite. Although infinite execution of a simulation may be useful in some cases, it is certainly not required in general. As described in Section 3.3, agent-based simulations are typically time-bounded.

A distinctive feature of agent-based simulations and a major difference to their non-simulation counterparts is that their dynamics are typically highly regulated and constrained. In a real-world simulation, this is achieved by some sort of *scheduling* mechanism which is either implemented manually or exists as part of the simulation framework being used (if any). In order to reflect this logic in an abstract way, we view both action ordering and termination as the tasks performed by a separate process which is composed in parallel with the $World$ process and forces it to act according to a certain scheduling strategy. The process that results from the composition of the $World$ and the scheduler process exhibits the ultimate behaviour of the simulation with all components performing their actions in the right order and the system terminating (i.e. blocking) after the final tick has been reached.

We distinguish between a *tick* and a *step*. A tick represents a logical and (from the perspective of an external observer) atomic period of time in which the entire population is updated. A step represents a single agent update only. The overall number of steps can thus be calculated by multiplying the number of ticks with the number of agents:

$$\#steps = \#ticks \cdot \#agents \tag{3.14}$$

When modelling a scheduler process, it is important to think about the *update mode*, i.e. the order in which things are supposed to happen. In the context of an agent-based simulation, we can generally distinguish between *synchronous* and *asynchronous* update. In a synchronous system, perception and update are two temporally separated steps. First, all agents perceive the environment. Since no state update happens during this step, the order in which agents perceive the task is entirely irrelevant. Agents can thus be seen as perceiving the environment *simultaneously*. After all agents perceived the environment, the state can be updated. Since, at this point, perception of the environment is complete, the order is again irrelevant. An important consequence of the synchronous update mode is that all agents perform their update based on the same environment state. Let $perceive.x$ and $update.x$ denote the perception and update action of agent x. For three agents, one tick could, for example, look as follows:

$$perceive.3 \rightarrow perceive.1 \rightarrow perceive.2 \rightarrow update.1 \rightarrow update.3 \rightarrow update.2 \tag{3.15}$$

In an asynchronous setting, each agent perceives the environment immediately before it performs its update. Perception and update are thus no longer temporally separated processes. Agent perform their perception and update steps sequentially, in an order which is determined by the scheduler. Different scenarios are possible: agents can be updated in fixed order (e.g. numerically or lexicographically sorted by their ids) or randomly. Since each agent perceives the environment shortly before it updates its state, the state it perceives is dependent upon its position in the update queue which may influence its behaviour. One possible interleaving of agent actions in an asynchronous setting is shown below:

$$perceive.3 \rightarrow update.3 \rightarrow perceive.1 \rightarrow update.1 \rightarrow perceive.2 \rightarrow update.2 \tag{3.16}$$

It has been shown for both cellular automata [65] and for agent-based simulations [47] that the choice of an update strategy has a significant influence on the dynamics of the system. The update mode also influences the nature of the system's state space and determines the way in which concurrently acting components are composed into an overall model on the transition system level [12].

Even though the asynchronous update mode can be found more commonly in the literature, both modes are described below. For asynchronous update, we further describe two particular ordering strategies: *fixed order* and *random order*. In order to keep the description of the formal framework as clear and modular as possible, the scheduling strategy is represented as a separate class which is composed in parallel with the population process Pop described above in order to restrict its traces accordingly.

All schedulers have in common that they need to know the maximum number of ticks in order to block the execution after the final tick. Furthermore, each scheduler can be in one of several control states which denote the currently available operation (*Perception* or *Update*). In order to formalise these commonalities, we make use of Object-Z's ability to model inheritance hierarchies and introduce a superclass $Scheduler$ which comprises one constant data field MAX_TICK denoting the total number of ticks in the simulation, a variable attribute $tick$ which denotes the current tick and a variable attribute $step$ denoting the current *step*. Remember that one of the tasks of the scheduler is to ensure the correct ordering of individual agent events. The $step$ variable ensures that actions are performed in the correct order, as described further below. The abstract $Scheduler$ class is shown below:

Scheduler

$$MAX\_TICK : \mathbb{N}$$

$$tick : \mathbb{N}$$
$$step : \mathbb{N}$$

Synchronous update: We start the description of the scheduler with the synchronous update mode. In synchronous update mode, all agents first perceive the environment before updating their state. As indicated above, the order of agent perception actions is irrelevant and is thus assumed to be random. Formally, random order update describes a strategy which, for each round, only allows the update of those agents that have not been updated before. For the description of the random order scheduler, the distinction between a $tick$ and a $step$ mentioned above becomes important. In the beginning of a *tick*, all agents are allowed to transition. One agent a_i is chosen and in the next *step*, all agents except a_i are allowed to transition, etc.

The logic of the synchronous scheduler is described by class $Scheduler_R$ which inherits from $Scheduler$ and further comprises two finite sets ids_1 and ids_2 of natural numbers which help to model the random update order of individual agents: one set for the 'ready' agents and one for the agents that have already

been updated. Once the former set is empty, a new round starts. Both the $tick$ and the $step$ variable are initially set to 1:

$SyncScheduler[Scheduler]$

$ids_1, ids_2 : \mathbb{F}\,\mathbb{N}$

INIT

$ids_1 \neq \varnothing$
$ids_2 = \varnothing$
$tick = 1$
$step = 1$

In order to control the behaviour of individual agents, the scheduler provides two operations, $Perceive$ and $Update$, which have the same interface as the equally named operations in the agent and environment classes and are thus able to synchronise. It is important to note that, further above, synchronisation was used to model *communication* between the agents and the environment. Here, synchronisation is used to *constrain* the behaviour of the $World$ process by coupling it with a scheduler process which it is forced to 'march in step' with. The logic of the $Perceive$ operation is described in the following schema. The constraint part describes the conditions that need to hold in order for the operation to be enabled.

$SyncScheduler[Scheduler]$

$Perceive$

$\Delta(step, ids_1, ids_2)$
$\_id? : \mathbb{N}$
$\_es? : EState$

$tick \leq MAX\_TICK$
$step = 1$
$\_id? \in ids_1$
$\#ids_1 = 1 \Rightarrow$
$\quad ids_1' = ids_2 \land ids_2' = \varnothing \land$
$\quad step' = 2$
$\#ids_1 > 1 \Rightarrow$
$\quad ids_1' = ids_1 \setminus \{\_id?\} \land ids_2' = ids_2 \cup \{\_id?\} \land step' = 1$

Let us briefly explain the logic of the $Perceive$ operation. First, in order for the operation to be enabled, a number of conditions need to be satisfied. First, the end of the simulation must not already have been

reached, as denoted by $tick \leq MAX\_TICK$; second, the scheduler needs to be in the correct control state, as denoted by variable $step$; and third, the id of the agent that wants to participate in the $Perceive$ operation needs to be in the set of not-yet-processed agents ids_1. This ensures that every agent is only allowed to perceive the environment once in each tick. The following lines update the content of the sets ids_1 and ids_2. The id of every agent who has participated in the $Perceive$ operation is moved from ids_1 to ids_2. If all agents have perceived the environment (and ids_1 is empty), the scheduler moves into the next control state.

Note that, due to the use of finite sets instead of sequences, the execution order of individual $Perceive$ agents is not fixed and any interleaving is possible. As described above, this models random choice and will also be used in the description of the random asynchronous update mode further below.

Similar to $Perceive$, the $Update$ operation is modelled in such a way that it is available as long as there are agents in the population which have not yet performed their update:

$SyncScheduler[Scheduler]$

$Update$

$$
\begin{array}{l}
\Delta(step, ids_1, ids_2) \\
\_id? : \mathbb{N} \\
\_as? : AState \\
\_es? : EState \\
\hline
tick \leq MAX\_TICK \\
step = 2 \\
\_id? \in ids_1 \\
\#ids_1 = 1 \Rightarrow \\
\quad ids_1' = ids_2 \land ids_2' = \varnothing \land \\
\quad step' = 3 \\
\#ids_1 > 1 \Rightarrow \\
\quad ids_1' = ids_1 \setminus \{\_id?\} \land ids_2' = ids_2 \cup \{\_id?\} \land step' = 2
\end{array}
$$

We assume that, once both stages (perception and update) have been finished for all agents, a tick is completed. In order to make the completion of a tick explicit, the scheduler performs a $Tick$ operation which has the same signature as the equally named operation in the $Environment$ class and comprises one out- (the current tick number) and one in-parameter (the state of the system). The operation schema for $Tick$ is shown below:

$SyncScheduler[Scheduler]$

$Tick$

$\Delta(tick, step)$
$\_tick! : \mathbb{N}$
$\_gstate? : GState$

$tick \leq MAX\_TICK$
$step = 3$
$\_tick! = tick$
$tick' = tick + 1$
$step' = 1$
$ids'_1 = ids_2$
$ids'_2 = \varnothing$

As described in the constraint part of the operation schema, $Tick$ is only available if the scheduler is in step 3, i.e. once both perception and update have been completed for all agents. $Tick$ possesses a single out-parameter containing the number of the finished tick which allows it to announce the completion externally. In addition to that, $Tick$ also resets the internal data structures in order to enable a new round of perception and update operations.

Analogous to all previously described classes, $SyncScheduler$ can be translated into a CSP process. The result is shown below.

$$
\begin{array}{l}
SyncScheduler = \\
\quad \text{let} \\
\qquad State_1(t, \{x\}) = perceive.x?\_ \rightarrow State_2(t, Ag) \\
\qquad State_1(t, xs) = perceive?x : xs?\_ \rightarrow State_1(t, xs \setminus \{x\}) \\
\qquad State_2(T, \{x\}) = \\
\qquad\quad \text{if } T == MAX\_TICK \text{ then} \\
\qquad\qquad update?x?\_?\_ \text{ then } tick.t?\_ \rightarrow STOP \\
\qquad\quad \text{else} \\
\qquad\qquad update?x?\_?\_ \text{ then } tick.t?\_ \rightarrow State_1(t+1, Ag) \\
\qquad State_2(t, xs) = update?x : xs?\_?\_ \rightarrow State_2(t, xs \setminus \{x\}) \\
\quad \text{within} \\
\qquad State_1(1, Ag)
\end{array}
$$

$SyncScheduler$ prescribes a clear order in which operations should be conducted. In the Object-Z specifications given above, this operational order is achieved by means of preconditioning the operations on the value of attribute $step$. We have thus decided to slightly deviate from the direct translation from Object-Z to CSP. In CSP, rather than using explicit attributes for that purpose, control states can be represented more elegantly by subdividing the overall process into parametrised sub-processes and

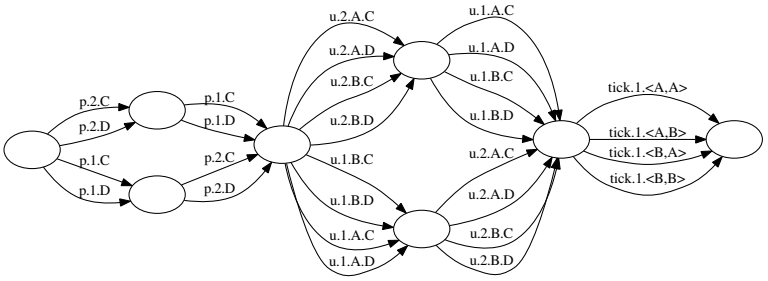


FIGURE 3.2: State space of $SyncScheduler$ for two agents, $AState = \{A, B\}$, $EState = \{C, D\}$ and $MAX\_TICK = 1$

pattern-match on their parameter values. In the CSP specification above, those sub-processes are named $State_1$ and $State_2$. Their first parameter t denotes the current time step, the second parameter denotes the number of agents yet to be updated. As shown in the last line, the second parameter is initialised with Ag, the set of all agent ids. The different sub-processes described between the 'let' and 'within' keywords are responsible for reorganising the second parameter, the set of yet-to-be processed agents.

A stylised version of the synchronous scheduler's state space for an agent-based simulation with $Ag = \{1, 2\}$, $ASTATE = \{A, B\}$ and $MAX\_TICK = 1$ is shown in Figure 3.2 (note that, for clarity, $perceive$ and $update$ have been renamed to p and u and some channel parameters are omitted). The graph makes the separation between perception and update stage apparent. First, all agents are allowed to perceive the environment in arbitrary order. After that, all agents are allowed to update their state, again in arbitrary order. After all agents have been processed, the scheduler sends a $tick$ event.

This concludes the definition of the synchronous scheduler class. We can now define the behaviour of a simulation with synchronous update as a parallel composition of $World$ and $SyncScheduler$, synchronising on the $perceive$, $update$ and the $tick$ events:

$$SyncABS = World \underset{perceive,update,tick}{\parallel} SyncScheduler \tag{3.17}$$

Asynchronous fixed order update: The asynchronous update mode assumes that perception and update are no longer actions which happen in separate global iterations such as in the synchronous case. Instead, each agent perceives the environment immediately prior to its update. As mentioned above, different orders of update are possible. In the fixed order scenario we assume there is an order which has been agreed upon in advance and which does not change during the runtime of the simulation. Since agents are often stored in an array, the update order may, for example, be determined by their position.

We model the fixed order scheduler as a class which comprises a sequence of agent ids that determines the order of updates:

$AsyncScheduler_F[Scheduler]$

$ids_1, ids_2 : \text{seq}\,\mathbb{N}$

INIT

$ids_1 \neq \langle\rangle$
$ids_2 = \langle\rangle$
$tick = 1$
$step = 1$

The operational interface of $AsyncScheduler_F$ is similar to that of $SyncScheduler$ and thus not described in further detail. The only difference lies in the way the internal control states are managed. Similar to the synchronous case, this logic can be found in the constraint part of the operations:

$ASyncScheduler_F[Scheduler]$

Perceive

$\Delta(step)$
$\_id? : \mathbb{N}$
$\_es? : EState$

$tick \leq MAX\_TICK$
$step = 1$
$step' = 2$
$\_id? = head(ids_1)$
$\#ids_1 = 1$

Update

$\Delta(step, order_1, order_2)$
$\_id? : \mathbb{N}$
$\_as? : AState$
$\_es? : EState$

$tick \leq MAX\_TICK$
$step = 2$
$step' = 3$
$\_id? = head(ids_1)$
$ids_1' = tail(ids_1)$
$ids_2' = \langle tail(ids_1) \rangle \frown ids_2$

Tick

$\Delta(tick, step)$
$\_tick! : \mathbb{N}$
$\_gstate? : GState$

$tick \leq MAX\_TICK$
$step = 3$
$ids_1 = \langle\rangle$
$step' = 1$
$\_tick! = tick$
$tick' = tick + 1$
$ids_1' = ids_2$
$ids_2' = \langle\rangle$

$AsyncScheduler_F$ can be translated into the following CSP process:

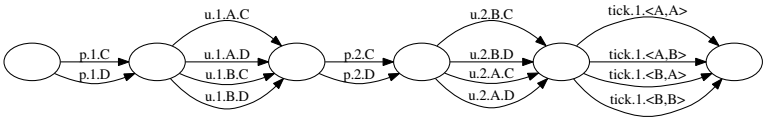


FIGURE 3.3: State space of the $AsyncScheduler_F$ process for two agents, $AState = \{A, B\}$, $EState = \{C, D\}$ and $MAX\_TICK = 1$

$$
\begin{array}{l}
AsyncScheduler_F = \\
\quad \text{let} \\
\qquad State(t, \langle x \rangle) = \\
\qquad\quad \text{if } t == MAX\_TICK \rightarrow \\
\qquad\qquad perceive.x?\_ \rightarrow update.x?\_?\_ \rightarrow tick.t \rightarrow STOP \\
\qquad\quad \text{else} \\
\qquad\qquad perceive.x?\_ \rightarrow update.x?\_?\_ \rightarrow tick.t \rightarrow State(t+1, seq(Ag)) \\
\qquad State(t, xs) = perceive.head(xs)?\_ \rightarrow update.head(xs)?\_?\_ \rightarrow State(t, tail(xs)) \\
\quad \text{within} \\
\qquad State(1, seq(Ag))
\end{array}
$$

A stylised version of the fixed asynchronous scheduler's state space for a simulation with $Ag = \{1, 2\}$, $ASTATE = \{1, 2\}$, $MAX\_TICK = 1$ and update order $\langle 1, 2 \rangle$ is shown in Figure 3.3. It becomes apparent that now, in asynchronous mode, perception and update are no longer separate processes for each agent. First, agent 1 perceives the environment, followed by its update. Then, agent 2 perceives the environment, followed by its update. Once both agents have been processed, the scheduler performs a $tick$ event.

A simulation with asynchronous fixed order update can now be described as a parallel composition of $World$ and $AsyncScheduler_F$, synchronising on the $perceive$, the $update$ and the $tick$ events:

$$AsyncABS_F = World \underset{perceive, update, tick}{\parallel} AsyncScheduler_F \tag{3.18}$$

Asynchronous random order update: In contrast to the fixed order case, random order update allows all possible interleavings of agent perception and update to occur. This idea has already been discussed in the synchronous case above and represented formally using sets of agent ids. This is also the only aspect which distinguishes class $AsyncScheduler_R$ representing the random order asynchronous scheduler from its fixed order counterpart $AsyncScheduler_F$ described above. The state of $AsyncScheduler_R$ is shown below:

$AsyncScheduler_R[Scheduler]$

$ids_1, ids_2 : \mathbb{F}\,\mathbb{N}$

INIT

$ids_1 \neq \varnothing$
$ids_2 = \varnothing$
$tick = 1$
$step = 1$

Function $Perceive$ is enabled once for each agent in each tick and directly followed by an agent's $Update$ operation:

$AsyncScheduler_R[Scheduler]$

$Perceive$

$\Delta(step)$
$\_id? : \mathbb{N}$
$\_es? : EState$

$tick \leq MAX\_TICK$
$step = 1$
$step' = 2$
$\_id? \in ids_1$

$Update$

$\Delta(step, ids_1, ids_2)$
$\_id? : \mathbb{N}$
$\_as? : AState$
$\_es? : EState$

$tick \leq MAX\_TICK$
$step = 2$
$\_id? \in order$
$ids_1' = ids_1 \setminus \{\_id?\}$
$ids_2' = ids_2 \cup \{\_id?\}$

Once all agents have performed their perception and update steps, operation $Tick$ is enabled:

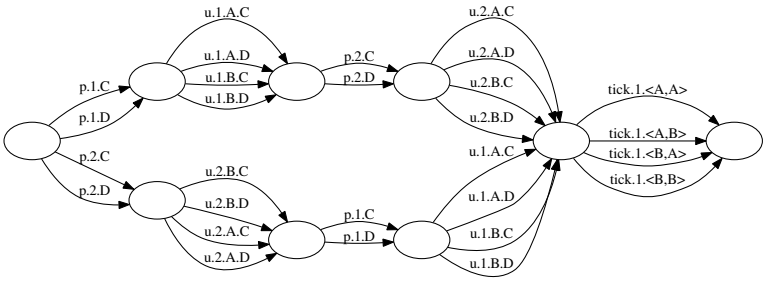


FIGURE 3.4: State space of the $AsyncScheduler_R$ process

$AsyncScheduler_R[Scheduler]$

Tick

$\Delta(tick, step)$
$\_tick! : \mathbb{N}$
$\_gstate? : GState$

$tick \leq MAX\_TICK$
$step = 3$
$ids_1 = \varnothing$
$step' = 1$
$\_tick! = tick$
$tick' = tick + 1$
$ids_1' = ids_2$
$ids_2' = \varnothing$

The class $AsyncScheduler_R$ corresponds to the following CSP process:

$$
\begin{aligned}
&AsyncScheduler_R = \\
&\quad \text{let} \\
&\qquad State(t, \{x\}) = \\
&\qquad\quad \text{if } t == MAX\_TICK \text{ then} \\
&\qquad\qquad perceive.x?\_ \rightarrow update.x?\_?\_ \rightarrow tick.t?\_ \rightarrow STOP \\
&\qquad\quad \text{else} \\
&\qquad\qquad perceive.x?\_ \rightarrow update.x?\_?\_ \rightarrow tick.t?\_ \rightarrow State(t+1, Ag) \\
&\qquad State(t, xs) = perceive?x : xs?\_ \rightarrow update.x?\_?\_ \rightarrow State(t, xs \setminus \{x\}) \\
&\quad \text{within} \\
&\qquad State(1, Ag)
\end{aligned}
$$

A stylised version of the random asynchronous scheduler's state space for a simulation with $Ag = \{1, 2\}$, $ASTATE = \{1, 2\}$ and $MAX\_TICK = 1$ is shown in Figure 3.4. It is visually apparent that

agents are allowed to be processed in arbitrary order, yet each agent performs perception and update successively. Once both agents have been processed, the scheduler sends a $tick$ event.

A simulation with asynchronous random order update can now be described as a parallel composition of $World$ and $AsyncScheduler_R$, synchronising on the $perceive$, $update$ and $tick$ events:

$$AsyncABS_R = World \underset{perceive,update,tick}{\parallel} AsyncScheduler_R \tag{3.19}$$

3.4.5 Examples

The purpose of this section is to show the correspondence between the abstract formal framework described above and two real-world examples; it is important to mention that this is not supposed to show the *practical applicability* of the framework, but rather its *descriptive representativity*. The first example is a simple social transmission model similar to the one introduced in Section 1.2.1; it is characterised by a small number of agents with low complexity. As a second example we chose the well-known Sugarscape model which has originally been published by Epstein and Axtell [89]. Both models differ significantly in both agent and system complexity as well as in their general structure and behavioural dynamics and therefore mirror well the variety of different types of simulation models.

In order to represent a concrete instance of an agent-based simulation, we can *refine* the abstract framework described above. This is done by (i) defining the domain of attribute names and values, and (ii) refining four functions which describe the model-specific logic in an abstract way. The descriptions in the following (particularly those for the second example) have been deliberately kept brief and focussed on the conceptual aspects of the refinement process. Also note that we follow an informal refinement process.

Example 1: Social transmission The first example is a simple social transmission model similar to the one introduced in Section 1.2.1. Instances of this type can be used to simulate a variety of social processes such as the spreading of diseases or information in human societies, the diffusion of computer viruses in networks as well as the adoption of new technologies or products in a society. Transmission models can be of arbitrary complexity, both in terms of agent state space and population size. For illustration, we keep the model deliberately simple without loss of generality.

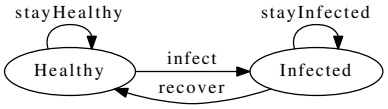


FIGURE 3.5: The behavioural transition system for agents in the disease transmission simulation

We define the population to consist of three agents each of which can be in one of two possible states: $healthy$ and $infected$. Each agent follows a simple set of rules:

1. If agent is infected:
   (a) If all neighbours are infected → stay infected
   (b) If one neighbour is infected and one is healthy → stay infected or recover (nondeterministic choice)
   (c) If all neighbours are healthy → recover
2. If agent is healthy:
   (a) If all neighbours are infected → become infected
   (b) If one neighbour is infected and one is healthy → stay healthy or infect (nondeterministic choice)
   (c) If all neighbours are healthy → stay healthy

The resulting state transition system for an individual agent is shown in Figure 3.5. The rule set comprises nondeterministic choices. In a realistic model, actions with multiple outcomes are typically associated with probabilities in order to describe their relative occurrence. For simplicity, we abstract away from the underlying probabilities and describe rules that yield multiple successor states as nondeterministic choices.

In order to formalise the logic of the disease transmission model, we first need to think about the sets $Name$ and $Value$ which represent attribute names and their values. In the disease transmission model, the only purpose of the environment is to enable agents to communicate with each other. To this end, the environment holds an attribute $popState$: seq $AState$ which holds the state of the population. The agents need an attribute $health : \{healthy, infected\}$ which describes their current infection state. Since an agent's rules are based upon the cardinality of neighbours being infected, it is not necessary for the agent to store the full information about the neighbourhood (i.e. which agent is in which state). Instead,

we assume that an agent perceives its environment in such a way that it only counts the number of infected and healthy agents. This is done by adding a symbolic constant $nbState : NBCounter$ where $NBCounter = \{0H2I, 1H1I, 2H0I\}$ to its perception store which represents a counter of the number of infected or healthy neighbours, respectively. For example, $2H0I$ denotes that two neighbours are healthy and no agents are infected.

The sets $Name$ and $Value$ can now be defined as follows:

$$Name = \{popState, health, nbState\}$$
$$Value = \mathbb{N} \cup \{healthy, infected\} \cup \{0H2I, 1H1I, 2H0I\}$$

The environment has no internal logic, so we can set the $update$ function of the $STEnvironment$ class to be the identity function:

STEnvironment

$$update : (EState \times GState) \rightarrow EState$$

$$\forall\, es : EState;\ gs : GState \bullet$$
$$update(es, gs) = es$$

The next part that needs to be defined is the set of capabilities, i.e. the set of actions that an agent is able to perform. As described above, actions are partial functions from the set of agent states to the set of agent states and correspond with state transitions. The transitions in Figure 3.5 can thus be translated into the following actions:

$STAgent$

$$infect : Action$$

$$\forall as : AState \bullet$$
$$as(health) = healthy \Rightarrow infect(as) = as \oplus \{health \mapsto infected\}$$

$$recover : Action$$

$$\forall as : AState \bullet$$
$$as(health) = infected \Rightarrow recover(as) = as \oplus \{health \mapsto healthy\}$$

$$stayInfected : Action$$

$$\forall as : AState \bullet$$
$$as(health) = infected \Rightarrow stayInfected(as) = as$$

$$stayHealthy : Action$$

$$\forall as : AState \bullet$$
$$as(health) = healthy \Rightarrow stayHealthy(as) = as$$

This results in the following set of capabilities:

$STAgent$

$$capabilities = \{infect, recover, stayInfected, stayHealthy\}$$

In the agent class, the following four functions need to be refined: $mergePerc$, $selectAction$, $actInt$ and $actExt$. The purpose of $mergePerc$ is to update the agent's attributes based on the environmental perceptions. To this end, we assume the existence of a helper function $count : EState \rightarrow NBCounter$ which translates the current state of the population into a 'neighbourhood infection counter' and whose logic shall not be further described here. $mergePerc$ can then be defined as follows:

$STAgent$

$$mergePerc : (AState \times EState) \rightarrow AState$$

$$\forall as : AState;\ es : EState \bullet$$
$$mergePerc(as, es) = attributes \oplus \{nbState \mapsto count(es)\}$$

The next function to be refined is $selectAction$ which the agent uses to select the next set of actions based on its current state:

$$
\begin{array}{|l}
\hline
STAgent \\
selectActions : (AState \times EState) \to \mathbb{F}\, Action \\
\hline
\forall\, as : AState;\ es : EState \bullet \\
\quad (as(health) = healthy \land as(nbState) = 2H0I \\
\qquad \land\ selectActions(as, es) = \{\{stayHealthy\}\}) \lor \\
\quad (as(health) = healthy \land as(nbState) = 1H1I \\
\qquad \land\ selectActions(as, es) = \{\{stayHealthy\}, \{infect\}\}) \lor \\
\quad (as(health) = healthy \land as(nbState) = 0H2I \\
\qquad \land\ selectActions(as, es) = \{\{infect\}\}) \lor \\
\quad (as(health) = infected \land as(nbState) = 2H0I \\
\qquad \land\ selectActions(as, es) = \{\{recover\}\}) \lor \\
\quad (as(health) = infected \land as(nbState) = 1H1I \\
\qquad \land\ selectActions(as, es) = \{\{stayInfected\}, \{recover\}\}) \lor \\
\quad (as(health) = infected \land as(nbState) = 0H2I \\
\qquad \land\ selectActions(as, es) = \{\{stayInfected\}\}) \\
\hline
\end{array}
$$

The next function to be refined is $actInt$ which updates the agent's state based on the selected set of actions:

$$
\begin{array}{|l}
\hline
STAgent \\
actInt : \mathbb{F}\, Action \to AState \\
\hline
\forall\, as : AState;\ acs : \mathbb{F}\, Action \bullet \\
\quad (as(health) = healthy \land acs = \{infect\} \\
\qquad \land\ actInt(acs, as) = as \oplus \{health \mapsto infected\}) \lor \\
\quad (as(health) = healthy \land acs = \{stayHealthy\} \\
\qquad \land\ actInt(acs, as) = as \oplus \{health \mapsto healthy\}) \lor \\
\quad (as(health) = infected \land acs = \{recover\} \\
\qquad \land\ actInt(acs, as) = as \oplus \{health \mapsto healthy\}) \lor \\
\quad (as(health) = infected \land acs = \{stayInfected\} \\
\qquad \land\ actInt(acs, as) = as \oplus \{health \mapsto infected\}) \\
\hline
\end{array}
$$

The final function to be refined is $actExt$ which is responsible for an agent's acting upon the environment. In the disease transmission example, the only purpose of the environment is to make the state of the population accessible to the agents. For that reason, function $actExt$ can be left undefined.

This concludes the specification of the disease transmission model. The chosen example was deliberately kept simple but, given the widespread usage of transmission models, it represents a typical class of agent-based models. A more sophisticated example including considerable agent and system complexity is described in the next section.

Example 2: Sugarscape As a second scenario we chose a simple version of Sugarscape, a popular agent-based model which has originally been published by Epstein and Axtell [89]. It is also part of the NetLogo model library [157, 240] and thus freely available for exploration.

For clarity, we keep the description of the Sugarscape model even more abstract than the disease transmission example in the previous section. Functions are thus only explicated verbally and not formalised in their entirety.

The agents in the Sugarscape model live in a two-dimensional grid world which has a dimension of 51×51 cells. Each grid cell contains a particular amount of sugar which can be consumed by the agent occupying the cell. In each tick, the agents look out for the closest cell with a sufficiently high amount of sugar and move there. Each movement consumes energy (defined by the agent's metabolism rate) which is increased again by consuming the sugar found in the cell. An agent dies if its energy level falls below a given threshold. The original model described by Epstein and Axtell also allows for further activities such as reproduction, information transfer, pollution, etc. For the sake of simplicity and clarity, however, we omit them in the following description without loss of generality and limit our description to the basic features (movement, food consumption and metabolism).

In contrast to the disease transmission example given above, agents are not connected to each other through a fixed topology and there is no direct communication. Instead, interactions happen indirectly, i.e. through changes to the spatial environment in which the agents are situated. We define a grid cell $Cell == \mathbb{N}_1 \times \mathbb{N}_1$ as a tuple of positive natural numbers. We also define the convenience type $Cells == \mathbb{F}\, Cell$ which denotes a finite set of cells. The environment itself comprises a set of cells, information about the dimensions of the space (number of cells in x and y direction), the amount of sugar per cell, the maximum possible amount of sugar and the Boolean occupancy state of each cell. The environment thus needs the following individual attributes, the union of which defines the overall set of attributes $attributes$:

SCEnvironment

$Boolean ::= True \mid False$

$$
\begin{aligned}
&width, height : \mathbb{N}_1 \\
&maxSugar : \mathbb{N}_1 \\
&cells : Cells \\
&sugar : Cells \to \mathbb{N} \\
&occupied : Cells \to \{True, False\}
\end{aligned}
$$

An agent in the Sugarscape model is an entity which moves through the spatial environment and eats grass in order to stay alive. We define the state of an agent as a set comprising its position, its sugar level and its metabolism rate:

SCAgent

$$pos, nextPos : Cell$$
$$sugar : \mathbb{N}_1$$
$$metabolism : \mathbb{N}_1$$

The sets $Name$ and $Value$ can now be defined as follows:

$$Name = \{width, height, maxSugar, cells, sugar, occupied, pos, nextPos, sugar, metabolism\}$$
$$Value = \mathbb{N} \cup Cells \cup (Cells \to \mathbb{N}) \cup (Cells \to \{True, False\}) \cup Cell$$

The initial state of the environment is determined by the assignment of values to (i) the environment's dimensions, (ii) the amount of sugar in each cell and (iii) the occupancy state of each cell. Clearly, there is a plethora of possible initialisations (actually as many possible initialisations as there are states in the state space). For the sake of brevity and clarity, we will not give a full description of the initialisation with all its redundant information but instead describe it schematically as follows:

SCEnvironment

INIT

$$width = 51$$
$$height = 51$$
$$maxSugarEnv = 10$$
$$sugar = \{(1,1) \mapsto 0, (1,2) \mapsto 1, ..., (51,51) \mapsto 10\}$$
$$occupied = \{(1,1) \mapsto False, (1,2) \mapsto True, ..., (51,51) \mapsto True\}$$

Let us now consider the logic of the environment. In our version of the Sugarscape model, grass will immediately grow back after it was consumed by an agent. We can view this as an autonomous action of the environment and describe it by means of the $update$ function which shall not be described in further detail here.

In order to define the set of capabilities, we need to think about the actions that an agent can perform. First, an agent needs to be able to move to a neighbouring cell. It also needs to be able to loose energy,

to consume sugar and to die. The capabilities of the agent is then defined as follows:

$$SCAgent$$
$$capabilities = \{loseEnergy, consumeSugar, die, stay, move\}$$

The first step of the agent behaviour comprises the perception of the environment. In the Sugarscape model, this first involves determining the set of neighbouring cells. An agent then needs to search for the optimal cell in its environment. The decision about the optimality of a cell is based on various criteria: visibility, occupancy state, highest level of sugar, and distance. The $mergePerc$ function of the agent class represents the sequence of actions necessary to translate the environment percepts into information about the optimal cell within the agent's neighbourhood and updates the agent's current state with it.

Based on the selected cell, the next set of actions can be chosen. Function $selectActions$ uses the information about the agent's current position together with the information about the optimal cell in the neighbourhood to determine which of the capabilities needs to be selected in order to change the agent's position accordingly.

After selecting the next action, the internal state of the agent as well as the state of the environment needs to be updated. It is debatable whether the update of an agent's position is internal or external. We believe it involves both: updating an agent's knowledge about its own position (internal) and the movement action (external). $actInternal$ thus represents the update of the agent's pos attribute as well as the adjustment of its sugar level based on the metabolism rate. $actExternal$ updates the environments information about cell occupancy and the level of sugar in the cell that the agent moves to.

This concludes the specification of the Sugarscape model. Despite its complexity, its logic can be described at an arbitrary level of abstraction by refining two data types and four functions of the abstract framework. In the description above, for example, we chose not to further explicate the functions which an agent uses to perceive its environment and select an optimal cell. Instead, we gave an abstract, high-level description in terms of relations between sets. This could be replaced by more comprehensive schemas which describe the functions' internal semantics in more detail in subsequent refinement steps until a level of granularity has been reached which corresponds directly with an implementation.

3.4.6 Summary

This concludes the description of the formal framework. It has been deliberately kept abstract in order to accommodate a large variety of different real-world models. Custom model-specific logic can be seen as being hidden in the sets $Name$ and $Value$ as well as behind the following four functions:

1. Environment:

   $update$**:** Updating the environment's state

2. Agent:

   $mergePerc$**:** merging the environment percepts into the current state

   $selectAction$**:** selecting a set of actions for execution

   $actInternal$**:** acting internally, i.e. updating the internal state

   $actExternal$**:** acting externally, i.e. changing the environment's state

Examples of how *refinement* of those functions and sets can be used to describe the logic of a concrete simulation model were given in Section 3.4.5.

Similar to the transition system semantics of the basic components given in Section 3.4.3, those for the macro level are described in the following section. In analogy to the notion of *agent traces* which represent sequences of agent states, the notion of *simulation traces* as sequences of group states is introduced and explained. In Section 3.5, the interface between an agent-based simulation and a monitor process is defined as a set of simulation traces. In Section 3.6, the formal framework is extended with probabilities. Furthermore, the section introduces the notion of *events* which forms the basis for the formulation of correctness properties in subsequent chapters.

3.5 Simulation traces

The purpose of the processes developed in the previous section is to provide a succinct high-level representation of an agent-based simulation as seen from the perspective of an external observer which is interested in the temporal evolution of the simulation's cognitive state. As described in the introduction, the goal of this chapter is to clearly define the interface between an arbitrary simulation and a runtime

monitor; this interface will then form the basis for the data structures upon which both properties will be formulated (see Chapters 5 and 7) and the actual verification will be performed (see Chapter 6). In this section we show that the three processes identified in the previous sections are equivalent to the abstract process DTS which we defined in Section 3.3 in terms of their externally observable behaviour and derive from it the notion of *simulation traces* as the central data structure that the monitor operates upon.

In the previous section, three different final versions of the simulation processes which differ only in how agent updates are ordered were developed:

$$SyncABS = World \underset{perceive,update}{\parallel} SyncScheduler \tag{3.20}$$

$$AsyncABS_F = World \underset{perceive,update}{\parallel} AsyncScheduler_F \tag{3.21}$$

$$AsyncABS_R = World \underset{perceive,update}{\parallel} AsyncScheduler_R \tag{3.22}$$

All of these processes start in an initial state and evolve over time. At any point in time, each of the processes is in a state which is itself composed of the states of the process' constituents. The most atomic processes are agents and the environment whose states are of type $AState$ and $EState$, respectively. We can now examine the evolution of the processes over time in more detail. However, as described above, since we are not interested in the agents' decision making process but instead view them from an 'external' point of view, we first need to reflect this fact on a process level by making their internal workings unobservable to the outside world. This can be achieved by hiding the internal $perceive$ and $update$ events:

$$HSyncABS = SyncABS \setminus \{perceive, update\} \tag{3.23}$$

$$HAsyncABS_F = AsyncABS_F \setminus \{perceive, update\} \tag{3.24}$$

$$HAsyncABS_R = AsyncABS_R \setminus \{perceive, update\} \tag{3.25}$$

We can now revisit process DTS introduced in Section 3.3 which we used as an 'external' representation of the full underlying agent-based simulation. The purpose of this process was to define the interface between the monitor and the simulation. DTS was defined as a simple process which regularly performs $tick$ events, as part of which the current global state of the simulation is emitted to an external observer.

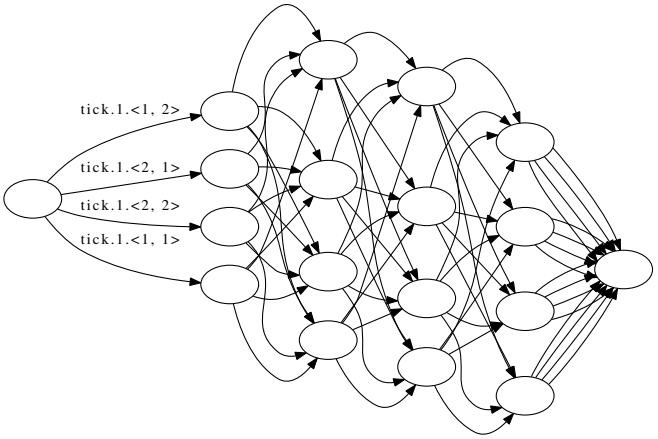


FIGURE 3.6: State space of the DTS process for 2 agents, $STATE = \{1, 2\}$ and $MAX\_TICK = 5$

An example of the state space of DTS for 2 agents, $STATE = \{1, 2\}$ and $MAX\_TICK = 5$ is shown in Figure 3.6 (for clarity, event labels are only shown for the first tick). It is apparent that the number of states of the process in each time step corresponds with the number of possible combinations of individual cognitive states (2^2 in this case).

We can now show that, to an external observer, the three simulation processes defined above are, in fact, equivalent to process DTS and thus satisfy our requirements. In order to formally prove that, we show that they are trace equivalent [211].

Proposition 1. Processes 3.26 - 3.28 are all trace-equivalent to process DTS.

Proof. The proposition can be proved showing that the following assertions are true using a refinement checker[5]:

$$DTS \equiv_{\mathrm{T}} HSyncSched \tag{3.26}$$

$$DTS \equiv_{\mathrm{T}} HAsyncSched_F \tag{3.27}$$

$$DTS \equiv_{\mathrm{T}} HAsyncSched_R \tag{3.28}$$

□

There is still a final step to be done. As described above, the processes have been deliberately kept general in order to accommodate a large variety of different types of simulations. To this end, any

[5] We used FDR3 to prove the assertions [210].

model-specific logic has been left out. As a consequence, no restrictions are imposed upon the possible transitions of the processes. This is the reason why all processes are trace-*equivalent* to DTS. It is obvious that, for any concrete refinement of the abstract processes (such as the ones described in Section 3.4.5), this equivalence will no longer hold. We show below that, no matter how complex the underlying model might be, the simulation processes are still *trace refinements* of process DTS. If process B is a trace refinement of process A (denoted $B \sqsubseteq_{\mathrm{T}} B$), then every trace of B is also a trace of A:

$$A \sqsubseteq_{\mathrm{T}} B \mathrel{\hat{=}} traces(B) \subseteq traces(A) \tag{3.29}$$

We make the following proposition:

Proposition 2. Any instance of one of Processes 3.23 - 3.25 is a trace refinement of process DTS.

Proof. Refining any of the processes 3.23 – 3.25 invariably involves the addition of transition rules. That is, instead of allowing an agent to transition into *any* successor state, rules are added which restrict the transition according to certain conditions and allow the agent to transition into a *subset* of successor states. For example, in the disease transmission simulation, an agent may transition from $susceptible$ to $infected$ if and only if a certain fraction of its neighbourhood is also in the $infected$ state. Let T denote the transition relation of process DTS. Further let ABS denote a process which represents a refinement of one of the processes 3.23 – 3.25. Let T' denote the transition relation of process ABS. Since T already contains all possible transitions, we can be sure that the following relation holds: $T' \subseteq T$. From that we can conclude that $traces(ABS) \subseteq traces(DTS)$ and also that $DTS \sqsubseteq_{\mathrm{T}} ABS$. □

Each agent-based simulation process in which internal agent events are hidden, regardless of the underlying scheduling mechanism, thus exhibits the same traces as the one shown in Figure 3.6. By agreeing on DTS as the 'least common denominator' for all three types of simulations, we can now define how the temporal evolution of a simulation appears from the point of view of an external observer. To an external observer (e.g. the monitor), the temporal progression of an agent-based simulation can be modelled as a set of *simulation traces*. We define the set Tr_s as the set of all possible simulation traces, i.e. sequences of group states:

$$Tr_s == \mathrm{seq}\ GState \tag{3.30}$$

Each individual simulation trace can be assembled at runtime by engaging with the simulation process and recording its repeatedly emitted global states. The notion of simulation traces provides a formalisation of a single simulation run from an external observer's point of view. It forms the basis for the definition of *events* (given in the following section), which themselves forms the basis for the formulation of *properties* further described in Chapter 4.

3.6 Events, properties, and their probabilities

When analysing the temporal dynamics of an agent-based simulation, one is typically interested in the occurrence of certain *events*, i.e. things that happen at some point during the execution or, in the case of a temporal event, over a period of time. Furthermore, due to the internal complexity of an agent-based simulation which arises from the individual components' nondeterministic decision making and the possibility of different interleavings because of random agent update order, the focus of interest during analysis is typically on the *probability* of certain events.

In the formal framework described above, the decision making process is described as a nondeterministic choice between possible successor states. From an operational point of view, this is not satisfying since nondeterminism cannot be resolved by a machine. If a certain situation may evolve into multiple possible successor situations, it is thus common to resolve nondeterminism by means of probability evaluations. One reason for using randomness is to represent uncertainty in the agent's decision making process. Especially when using a utility-based approach, probabilities represent a convenient way to model the prioritisation of certain actions over others. Another typical reason for the use of randomness is the derivation of an agent-based model from existing data using, for example, machine learning techniques. In this case, agents are often represented as probabilistic processes whose behaviour is purely described by stochastic state transitions which can be seen as numeric abstractions of the underlying decision making process.

The formal framework described above does not incorporate probabilistic state transitions explicitly. This is due to the fact that neither Object-Z nor CSP has built-in support for probabilities. Instead, the decision making process is represented as a nondeterministic choice. It is important to note that probabilities are not strictly necessary to model the temporal progression of the internal state of a simulation and to prove basic temporal properties. It is perfectly possible to formulate and prove reachability, safety and liveness conditions without reverting to probabilities. If statements about the *probability* of

something happening need to be made, however, transition probabilities need to be integrated into the formal specification.

The purpose of this section is to formally associate the set of traces of an agent-based simulation as developed in Section 3.5 with a *probability space*. This allows us to talk about events and their probabilities. We show that, by varying the set of outcomes that one focusses on, events of different granularity become detectable.

3.6.1 A brief excursus on probability theory

If agent decisions are probabilistic, then the transitions between states in the resulting state space are also probabilistic. Let $P : GState \times GState \rightarrow [0, 1]$ be a *probabilistic transition function* such that $\forall\, s : GState \bullet \sum_{s' \in GState} P(s, s') = 1$. This results in the representation of the simulation dynamics as a *probabilistic transition system* in which each state contains all the information that is necessary to determine the next steps — a *Markov chain*. It is important to note that this formal representation serves a merely illustrative purpose here; due to its focus on individual traces produced by a running simulation, our framework does not require the presence of an explicit Markov chain representation for verification. However, in order to understand how the notion of probability against the background of individual traces is defined in our work, it is helpful to start with such a formal description.

In the presence of transition probabilities, it is intuitively clear that each trace $\omega = \langle s_0, s_1, ..., s_n \rangle$ starting from the initial state has a certain probability of occurring, denoted $Pr(\omega)$, which is the product of all individual transition probabilities:

$$Pr(\omega) = P(s_0, s_1) \cdot P(s_1, s_2) \cdot ... \cdot P(s_{n-1}, s_n) \tag{3.31}$$

$$= \prod_{0 \leq i < n} P(s_i, s_{i+1}) \tag{3.32}$$

In the presence of long simulation runs, restricting the focus of attention to the probability of full traces may be too coarse-grained. Traces represent (possibly long) sequences of system states which themselves also have a complex internal structure; in the course of a simulation run, numerous *events* take place which constitute themselves as changes to the state of the system. A trace represent all the states of the underlying run and can thus be seen as a rich source of analysis. As described in Section 2.2.3, the purpose of internal validation is to assess the correctness of causal mechanisms within the model.

According to the probabilistic theory of causation [124], events are causally related if the potential cause raises the probability of the potential effect. In order to answer those types of question, a fine-grained analysis of events and their probabilities is important. In addition to the probability of the trace itself, it is therefore useful to also determine the probability of all individual events represented by it. In order to talk about events and their probability, we first need to make sets of traces *measurable*. To this end, we associate a *probability space* with the set of simulation traces, a process which is described below. For a comprehensive overview of probability theory in general and its relation to the verification of probabilistic systems in particular, please refer to the relevant literature [191, 12].

A probability space is a triple (Ω, Σ, Pr) where Ω is the *sample space*, $\Sigma \subseteq \mathbb{P}\,\Omega$ is a σ-algebra and $Pr : \Sigma \to [0, 1]$ is a probability measure. The sample space Ω can be seen as the set representing all possible *outcomes* of an experiment. Imagine, for example, throwing a die. In this case, the sample space is $\Omega = \{1, 2, 3, 4, 5, 6\}$. We can now start to define possible events within the set of outcomes. A single event represents a set of outcomes which all satisfy a common criterion. For example, getting an even number when throwing a die is represented by the set $\{2, 4, 6\}$. Formally, the set of events forms a *σ-algebra* $\Sigma \subseteq \mathbb{P}\,\Omega$ on Ω, where Σ is a subset of the power set of Ω. A σ-algebra Σ also needs to satisfy the following requirements:

- Σ contains the empty set $\varnothing$
- Σ is closed under complements: if A is in Σ then so is its complement $\overline{A} = (\Omega \backslash A)$
- Σ is closed under countable unions: if $A_1, A_2, ...$ are in Σ then so is their union $A = \bigcup A_n$

Furthermore, in order to assign a probability with an event, we need a *probability measure* $Pr : \Sigma \to [0, 1]$ which is a function that assigns to each event $E \in \Sigma$ a number between 0 and 1. Pr also needs to satisfy the following requirements:

- Pr is countably additive: for all countable collections $A = \{A_1, A_2, ..., A_n\} \in \Sigma$, $Pr(\bigcup A_i) = \sum(Pr(A_i))$
- $Pr(\varnothing) = 0$ and $Pr(\Omega) = 1$

A probability space is then defined as a triple (Ω, Σ, Pr) comprising the sample space Ω, σ-algebra Σ and probability measure Pr. In the context of probability theory, the events $\omega \in \Sigma$ are said to be *measurable*.

3.6.2 Simulation and sampling

In order to talk about events in the context of simulation runs, the set Tr_s of simulation traces which a simulation can produce needs to be made measurable by associating it with a probability space. We start with the sample space. Since we assume that each simulation is time-bounded, all traces $\omega : Tr_s$ are finite. A trace obtained through a single simulation run (if properly randomised, which we assume here) can then be seen as a single sample drawn from the *set of finite traces* as defined by the logic within the model. However, at the same time, a single trace of length k also represents a set of k samples drawn from the *set of states* defined by the model. Furthermore, it also represents a set of $\binom{k}{2}$ samples drawn from the *set of state tuples*, a set of $\binom{k}{3}$ samples drawn from the *set of state triples*, and so on. Even more, given n agents, each simulation trace also represents n samples drawn from the *set of agent traces*, each of which itself represents a set of k samples from the *set of agent states*, etc.

In general, the description of a probabilistic state-based model yields a large range of different sets of outcomes that one can draw from: a set of agent or group states (one for each possible group of agents), of agent or group state pairs, of agent or group state triples, etc. Each individual simulation run represents one or many samples from each of those sets. As described above, each set of outcomes corresponds with a different probability space and thus allows for the detection of different events. As a consequence, just by interpreting the same outcome in different ways, different types of events become detectable.

It becomes clear that a single simulation trace already represents a rich source of analysis. Let us now briefly look at the types of outcomes that one is *typically* interested in. We can assume that, in a simulation context, we are mostly interested in events defined over *coherent trace fragments*, rather than over arbitrary tuples of states. Informally, a coherent trace fragment is any sequence of states which exists in the underlying state space. Fragments of length 1 represent individual states, fragments of length 2 represent states and their direct successors, fragments of length 3 represent states and their two subsequent states, etc. Formally, the set $\mathcal{F}_k$ of coherent trace fragments of length k is defined as the set of sub-sequences of states, i.e. sub-traces, of length k:

$$\mathcal{F}_k == \bigcup_{\omega \in Tr_s} \{p \,\mathrm{in}\, \omega \mid \#p = k\} \tag{3.33}$$

Each fragment size represents a certain level of *granularity* with respect to the simulation outcome. Before defining the sample space of a simulation, it is therefore important to clarify the granularity necessary to answer a given question. For example, some questions are formulated over entire simulation traces, i.e. members of the set $\mathcal{F}_t$. Typical representatives of this group are temporal questions that involve statements like, for example, *eventually* or *always*. In this case, the set from which samples need to be drawn is the set of all full traces, i.e. $\Omega = Tr_s$. The σ-algebra Σ (the set of possible events defined as a subset of the power set of Ω) thus represents the set of all possible sets of traces.

For other questions, a finer level of granularity is needed. Consider, for example, a question about the existence of a particular state transition. On a full simulation trace, the state transition of interest may occur several times. In order to detect all occurrences (and thus measure the event's probability), it is not sufficient to look at complete traces. Instead, we need to look at *trace fragments* of length 2, i.e. at tuples of immediately succeeding states drawn from the set $\mathcal{F}_2$. This is necessary since any state transition is described by its start and end state. If questions about the probability of a single agent attribute valuation are to be answered, i.e. questions about a particular property of an individual states, then the set which samples need to be drawn from is the set of trace fragments of length 1, i.e. the set of individual states.

We can generalise that, in order to answer any question, we need trace fragments of length k where $0 < k \leq t$ and t is the maximum number of time steps in the simulation. The sample space is then defined as the set of all fragments of length k, i.e. we have $\Omega = \mathcal{F}_k$. The σ-algebra Σ is a subset of the power set of Ω and thus represents the set of all possible sets of trace fragments of length k, i.e. $\Sigma \subseteq \mathbb{P}\,\mathcal{F}_k$.

In order to define a probability measure for any event in Σ, we first need to define the probability of a certain trace fragment. The probability of fragment $f = \langle s_j, s_{j+1}, ..., s_k \rangle$ of trace $t = \langle s_0, s_1, ..., s_n \rangle$ where $0 \leq j \leq n$ and $j \leq k \leq n$ is the probability of trace t divided by the number of coherent fragments of t of size $(k - j)$:

$$Pr(\langle s_j, ...s_k \rangle) = \frac{\prod_{0 \leq i < n} P(s_i, s_{i+1})}{n - (k - j) + 1} \tag{3.34}$$

The probability measure for any event $\sigma \in \Sigma$ (which represents a set of trace fragments) can then be defined as the sum of the probabilities of each trace fragment $\omega \in \sigma$:

$$Pr(\sigma) = \sum_{\omega \in \sigma} Pr(\omega) \tag{3.35}$$

The association of a probability space with a simulation transition system makes it possible to talk about events — things that happen within the universe generated by the simulation — and their probability. Events are described by *properties*. A property refers to a set of possible outcomes of a simulation. Consider, for example, a property φ which states that the system will eventually reach a given state s. This clearly requires to be answered on *full simulation traces*. The set of outcomes thus needs to be defined as the set of all trace fragments of length t where t is the maximum trace length. $\Sigma = \{\sigma \in \mathcal{F}_t \mid \sigma \models \varphi\}$ is then defined as the set of those trace fragments σ which satisfy this condition (denoted $\sigma \models \varphi$) and thus eventually end up in state s.

On the other hand, let ψ denote a property which states that the system transitions from a state in which n agents are infected to a state in which m agents are infected. Again, this property represents a property about the full population. However, due to its focus on transitions, it requires trace fragments of length 2 in order to be answered correctly, i.e. we have $\Sigma = \{\sigma \in \mathcal{F}_2 \mid \sigma \models \psi\}$.

As a final example, let ψ denote a property which states that *a single agent* transitions from an infected to a non-infected state. It describes a state transition and thus requires trace fragments of length 2, similar to the previous property. However, it is also of individual nature, i.e. the set of outcomes that it refers to is the set of all fragments of length 2 *of individual agent traces*. By formulating the properties in the appropriate way, we can answer quantitative properties about the behaviour of the full population, about the behaviour of groups within the population or about the behaviour of individual agents. By evaluating an individual property on independent and randomly chosen agent traces, we can even answer questions about the *expected* or *average behaviour* of individual agents. In Chapter 6, we present algorithmic procedures to perform the sampling based on full simulation traces.

Since, as described above, the traces of a simulation are measurable, the probability of any property ϕ is defined as the sum of the probabilities of all trace fragments of length k in the associated σ-algebra $\Sigma = \{\sigma \in \mathcal{F}_k \mid \sigma \models \phi\}$:

$$Pr(\phi) = \sum_{\sigma \in \Sigma} Pr(\sigma) \tag{3.36}$$

In order to make clear what fragment size a property is being verified upon (and thus, which interpretation of the sample space is being chosen), we add the sample size as a subscript variable to Pr. For example, we refer to the probability of a property ϕ which is to be evaluated upon trace fragments of size 2 as $Pr_2(\phi)$. We omit the subscript variable if (i) the formula is to be evaluated upon sets of full traces, i.e. if k is the length of the simulation traces obtained, or, (ii) if the fragment size does not matter for the purpose of description.

This concludes the description of events, properties and their probability in an abstract way. Let us briefly summarise the ideas described above. Essentially, a simulation trace, i.e. the output of a single simulation run, can be seen as a single sample from the set of finite traces defined by the model. Following this interpretation, the set of outcomes, i.e. the set which events and thus also properties are being formulated upon, is fixed as the set of finite traces. However, a single simulation trace can also be interpreted as a set of sample states drawn from the set of states, as a set of sample state tuples drawn from the set of state tuples, as a set of sample state triples drawn from the set of state triples, and so forth. Furthermore, sampling can be performed on the macro, meso and micro level and thus refer to the behaviour of the population, of groups of agents or of individual agents. Depending on how the set of possible outcomes is interpreted, different events can be defined which, ultimately, allows for the expression of richer properties. Given a property ϕ, its meaning and, of course, also its probability may vary depending on which set of outcomes it is interpreted on. It is therefore important to make the fragment size a central parameter of the verification algorithm.

An important aspect of this work is to analyse in more detail what types of properties are required for the verification of agent-based simulations and — in order for them to be checkable in an automated way — how they can be formulated in a mathematically rigorous way by means of linear temporal logic. This section laid the ground for an advanced discussion about outcomes, events and properties which is given in the next chapter.

3.7 Summary

Despite ever-increasing adoption of the agent-based modelling paradigm for the simulation of complex adaptive systems, attempts to formalise the underlying concepts are still largely missing. The content of this chapter was an attempt to close this gap by providing *formal specifications* of entities commonly

found in agent-based simulations with the overall purpose of defining a *clear formal interface between a simulation and an external runtime verification tool*.

Starting with an informal discussion of some typical characteristics of agent-based models and consequential design decisions for the formal framework in Section 3.3, the formalisation of basic components (the environment and the agent) was given in Sections 3.4.1 and 3.4.2. In order to keep the formalisation aligned with the object-oriented paradigm which is commonly employed in developing agent-based simulations, yet without losing formal rigour, specifications of non-composite entities were given in Object-Z. The formalisation continued with the description of the system level in Section 3.4.4. In order to keep the description succinct and focussed on their *interaction*, all entities were translated into the process algebra CSP.

In order to show its representativeness of models with different characteristics, two well-known examples from the agent-based modelling domain — a disease transmission model and a simple version of Sugarscape — were formulated in the language of the framework in Section 3.4.5. The first example comprised a simple network of fully connected agents. Each agent changes its state in a reactive manner purely based on the state of the neighbourhood. Communication is direct and there is neither a global environment nor any shared resources that can be accessed by the agents. The first example exhibited low agent and low system complexity. The second example represented a significantly more complex scenario and described a typical agent-based simulation scenario where agents are located in a spatial environment. Interaction happens through access to shared resources such as food and space within the environment. Both examples showed how model-specific logic could be represented by *refining* previously abstractly specified data structures and functions without compromising the overall structure of the framework.

The purpose of the remaining sections was to derive insights into the underlying transition system semantics from the formal framework by first introducing the notion of *simulation traces* as sequences of group states in Section 3.5. Based on that, the concept of *events and their probability* was introduced in Section 3.6. We showed that, by interpreting the set of outcomes produced by a simulation in different ways, *different levels of granularity* with respect to the detection of events can be achieved. Events form the basis for the formulation of *properties* which are themselves formulated over fragments of simulation traces.

The purpose of the next chapter is to elaborate on the notion of properties and discuss informally which types of validation questions one typically wants to formulate over the temporal dynamics of individual

simulation traces. To this end, the internal structure of simulation traces and the meaning of their constituents is discussed first. This serves as the basis for the identification of requirements that a formal language needs to satisfy in order to be able to express those validation questions effectively.

Chapter 4

Analysing individual simulation runs

4.1 Introduction

This chapter discusses requirements for the analysis of individual simulation runs against the background of a runtime verification framework. As mentioned in the previous chapter, a run can be formally represented as a *simulation trace* which denotes a sequence of global states. Each global state can be further subdivided into a set of individual or local states — one for each agent. Simulation traces can be obtained and made available to the runtime verification framework in a number of different ways. First, they can be produced by an existing simulation model (e.g. developed in NetLogo or Repast), by repeatedly executing the simulation and outputting the population state once every tick to a file or database. The so produced traces can then be read by the verification framework and used to answer correctness properties *a posteriori*, i.e. after the simulation has been executed. We refer to this analysis mode as *offline verification*. Second, traces can be produced and made available for verification by *connecting* the runtime verification framework to an existing simulation model and *requesting* states *on demand*, i.e. whenever required for verification. In this setting, the underlying simulation needs to provide a means for an external caller (i) to trigger the update of the population state (e.g. by means of an external `step` function), and (ii) to obtain the new state of the population in the format of a group state as formally described in Section 3.4.1 (e.g. as a return value of the `step` function). NetLogo, for example, can be executed in 'headless' mode (without user interface) and integrated into existing Java application. In that way, an existing simulation model can be controlled externally as required for verification. Due to the intertwining between simulation and verification, we refer to this mode as *online*

verification. And third, in order to integrate simulation and verification even more seamlessly, the logic of the simulation model to be verified can be formulated and executed directly within the *simulator* that forms part of MC^{2}MABS, the runtime verification framework developed as part of this work (see Chapter 8). The simulator is written in C++ and designed for high performance and efficient collaboration with the monitor; due to the tight coupling between simulator and monitor, this mode reduces the amount of housekeeping code (e.g. starting an external simulation tool, advancing the simulation, requesting a state, conversion, etc.) to a minimum and achieves best performance with respect to verification.

In this chapter, we are entirely agnostic about the actual way in which traces are produced and focus on their analysis irrespective of their origin. Given the typically vast number of possible traces that a given model can exhibit, the informative value of a trace may be rather small. Nevertheless, being able to analyse individual traces efficiently is critical; by analysing a sufficiently large number of individual runs, extensive insight into the general dynamics of the simulation can be obtained. This makes it possible to formulate statements about the overall behaviour of the agent-based simulation without having to search through the entire space.

Despite its limited explanatory power, a single simulation trace provides a rich source for formal analysis due to its hierarchical structure. It represents a complete history of the entire population which includes information about all of its subgroups as well as about every individual agent. The presence of a large number of concurrently acting agents clearly requires a high level of expressivity in a property specification language. For example, properties can be formulated over the entire population, over subgroups, over individual agents or over arbitrary combinations thereof. The overall goal of this chapter is to clarify those requirements and to identify the features that a formal language for the specification of simulation-related correctness criteria needs to provide.

We follow a bottom-up approach and define the requirements in an incremental way by first identifying the basic building blocks that a property specification language needs to provide — *atomic* and *composite observables* as well as *events* (which were introduced formally in the previous chapter) — in Sections 4.2. This is followed by an overview of different *property types* that are typically formulated about the temporal behaviour of agent-based simulations in Section 4.3. The chapter concludes with a summary in Section 4.5.

The character of this chapter is rather informal. Its purpose is to collect requirements for the design of a logic-based property specification language for agent-based simulations. Syntax and semantics of the language are described in further detail in the next chapter.

4.2 Building blocks

The identification of requirements for a formal specification language for correctness criteria is best done from the bottom up by first defining the individual *observables* that one is typically interested in during the analysis of an agent-based simulation. Atomic observables are determined by the nature of the *model* that the analysis is performed upon. In other words, we can only observe things which are reflected in the model that we are operating on. In the context of runtime verification, the model itself is defined by the *interface* between the simulation and the monitor. As described in Section 3.5, the formal model that we propose for state-based analysis is a set of *simulation traces* which represent sequences of global system states which themselves represent functions from the set of agent ids to the set of agent states:

$$\begin{aligned} Tr_s &== \text{seq } GState \\ &== \text{seq}(Ag \rightarrow AState) \\ &== \text{seq}(Ag \rightarrow (Name \rightarrow Value)) \end{aligned} \tag{4.1}$$

Due to the multi-level nature of an agent-based simulation, observables may relate to different observational levels (micro, meso and macro) and, for example, result from an *aggregation* process. Aggregations are important tools for complexity reduction when dealing with large groups of individuals and thus need to be supported in the language. Technically, aggregated values can be seen as *higher order* or *composite observables* which are themselves composed of atomic or composite observables. Finally, we need support for *events*, i.e. things that happen during a run and whose occurrence a correctness criterion is supposed to describe. The notions of observables, aggregations and events are described in further detail in the following sections.

4.2.1 Observables

We denote entities that represent the atomic points of interest during the analysis and form the building blocks around which complex temporal properties are constructed as *observables*. Parunak defines observables as "measurable characteristics of interest. They may be associated either with separate individuals (e.g., the velocity of individual gas particles in a box) or with the collection of individuals as a whole (the pressure in the box). In general, the values of these observables change over time. In both

[agent-based and equation-based] models, these observables are represented as variables that take on assignments" [196]. As described in the previous chapter, an agent's state is defined as the union of its knowledge, goals, memory etc. at any point in time. Since we are not concerned with the internals of an agent's decision making process, we abstract away from any mentalistic notion and simply refer to each value that constitutes the state as an *attribute* (see Chapter 3) which is defined as a simple key value pair $Att == (Name \times Value)$. Since every arbitrarily complex property of an individual is ultimately built around one or several individual agent attributes, we view the value of an agent attribute as the most atomic observable. Given a simulation trace $\omega : Tr_s$, the value of attribute i of agent a at time t is directly accessible as $\omega[t][a][i]$.

Requirement 1. A language for the formulation of correctness criteria needs to provide support for accessing atomic observables, i.e. agent attributes and their values.

In order to cope with combinatorial explosion, it is often necessary to step up one level of abstraction in order to simplify the system under consideration and focus on those aspects that one is most interested in. For example, when analysing the behaviour of an agent population on the macro level, one is typically not interested in a particular attribute of a particular agent at a particular time. Instead, one rather wants to study the *aggregate* behaviour of the system over time (the average income of a population is often more expressive than a list of 10,000 individual incomes). This is not to say, however, that information about the outliers (e.g. minimum and maximum income) or the study of individual agent traces is not interesting — quite the contrary; after all, being able to observe the behaviour of a complex system on multiple levels — micro, meso and macro — is one of the distinctive features of agent-based modelling. But for many scenarios, information about the aggregate behaviour is more meaningful than individual data — particularly if the system under consideration exhibits emergent behaviour which is by definition not reducible to the behaviour of its constituents.

Transforming multiple individual values into a single value creates a *higher order* or *composite observable*. The resulting value can be classified as an observable since it is an individual value that can be used in more complex statements; it can also be classified as being composite because every such transformation, regardless how nested it is, is composed of smaller building blocks, the most atomic ones of which are agent attributes and their values. On the micro level, aggregation can help to reduce a potentially huge number of states by constraining the focus of interest to a particular aspect of the agent's structure. For example, in a simulation where an agent has multiple sources of income, one might, for analysis purposes, only be interested in its *total income*. However, combinatorial explosion is particularly critical on the macro level. Due to exponential growth resulting from the parallel composition of individual

agent state spaces, the size of the population state space quickly becomes unmanageable — even for systems where the agents are of low individual complexity. In this case, aggregation helps to focus on the relevant aspects. Examples of aggregations on the macro level are manifold: we are interested in the *average wealth* of a population segment, the *maximum income* of a profession, the *minimum time* needed to finish a given task, etc. Aggregations can also be arbitrarily nested. Consider, for example, the case in which the average income of an individual agent (assuming that the agent has multiple jobs) is further aggregated into an average income for the entire population (or a segment thereof). Another important numerical aggregation method is *counting*. Given the high number of individual components in an agent-based simulation, many common questions about its global, emergent behaviour can be framed as a counting problem: we are interested in the *number of agents* being infected, the *number of purchases* of a given product or the *number of trades* in a particular timespan, etc.

Due to their composite nature, aggregation attributes are not directly accessible via the underlying simulation trace. However, rather than providing language constructs to allow for their formulation directly within the specification language, we recommend that their calculation be done externally and the resulting value be accessible by a symbolic name similar to the case of atomic observables described above. This is motivated by the assumption that a property specification language should provide a good balance between expressivity and simplicity. A language which is too simple may be useless for expressing certain properties whereas an overloaded language may be too hard to use. In order to access aggregate values by name, we therefore assume the existence of an external *aggregation function* on both the agent and the group level which is represented formally as a function from the set of attribute names and the set of agent or group states to the set of attribute values:

$$A_a == Name \times AState \rightarrow Value \tag{4.2}$$

$$A_g == Name \times GState \rightarrow Value \tag{4.3}$$

The way in which the actual aggregation takes place is not further specified. All the specification language then needs to provide is language constructs to facilitate the access to these aggregations.

Requirement 2. A language for the formulation of correctness criteria needs to provide support for accessing composite observables, i.e. values resulting from aggregation operations on both the agent and the group level.

Atomic and composite observables form the building blocks for the definition of *events* which are described in the following section.

4.2.2 Events

Section 3.6 in the previous chapter described events as sets of fragments of traces which all satisfy a common criterion. In this section, we investigate further the nature of those criteria against the background of single simulation traces and discuss what different types of events can be distinguished.

Complex adaptive systems are characterised by two central aspects: their ability to produce emergent behaviour and the ability of their constituents to adapt to changes in their environment. Arthur describes complex adaptive systems as systems in which "individual behaviors collectively create an aggregate outcome; and they react to this outcome" [6]; the macro level behaviour is influenced by the micro level behaviour and vice versa. Feedback loops are at the heart of complex systems and the reason for them to exhibit complex and emergent phenomena. On the other hand, they are also the reason why complex systems are notoriously hard to understand and to control. In order to analyse the dynamics of an agent-based simulation for the purpose of verification and validation in an automated way, expectations about the occurrence of certain events need to be formulated. But what are the types of events that one wants to observe on both micro and macro level? In order to answer this question, we distinguish between *simple* and *complex events*. The idea of simple and complex events is not new and has, for example, been discussed by Chen *et al.* [51] who use it to describe emergent phenomena in agent-based simulations in a formal way. According to them, events are recursive composite entities whose atomic building blocks are the observables described above (i.e. individual agent attributes or their aggregations). Chen *et al.* define an event as "a state transition defined at a particular level of abstraction" and distinguish between *simple (SE)* and *complex events (CE)*. Simple events are defined as particular state transitions occurring in time with a non-negative duration. Complex events are defined recursively as composite structures that can either represent simple events or a temporal relationship with optional additional constraints between complex events. Possible temporal relationships include, for example, *same time*, *before*, *after* or *immediately after*. Additional constraints include *spatial constraints* and *component* or *variable constraints*. Whereas the former type describes spatial relations between complex events, component or variable constraints describe relations between variables of complex events, e.g. arithmetic relations.

We adopt a slightly different view and describe a simple event as *a state of affairs which relates to a single system state*. This is motivated by the definition of events as sets of outcomes which all satisfy

a common criterion, as described in Section 3.6. In the simplest scenario, the outcomes represent individual agent attributes and their values. According to the definition of Chen *et al.*, state transitions are considered simple events. We believe that this definition is too coarse-grained and define a simple event as *any Boolean expression that makes a statement about (atomic or composite) observables and whose correctness can be checked on a single state*. Examples of simple events are the assignment of a particular value to an attribute, a particular logical combination of attribute values or an arithmetic relation between values[1].

As mentioned above, it is important to find a good balance between expressivity and simplicity of the property specification language. In order not to overload the language, we thus recommend to include only Boolean and arithmetic relations directly into the language and to represent any other simple event as *predicates* or *labels* which are defined externally and which can be referenced from within the language. Similar to aggregations above, we distinguish between *agent* and *group predicates* and assume that they can be accessed via dedicated *predicate functions*. Formally, an agent predicate function P_a is a n-ary function which accepts a predicate name, an agent state and a finite set of arguments and returns a Boolean value. A group predicate function can be defined accordingly:

$$P_a : Name \times AState \times \mathbb{F}\ Value \rightarrow \{\text{true}, \text{false}\} \tag{4.4}$$

$$P_g : Name \times GState \times \mathbb{F}\ Value \rightarrow \{\text{true}, \text{false}\} \tag{4.5}$$

Requirement 3. In order to support the formulation of simple events, a correctness criterion language needs to provide support for the description of particular values of observables, combinations of observable values (logical or arithmetic) and predicates on both the agent and the group level.

We further define a complex event recursively as the occurrence of a simple event or a temporal relationship between complex events. As opposed to simple events whose correctness is evaluated on trace fragments of length 1 (i.e. individual states), complex events are evaluated on trace fragments with length k where $k \geq 2$. Consider again the transmission model introduced in Section 1.2.1. In this case, we could, for example, be interested in the stability of the population with respect to the infection state; we want to make sure that, starting from an initial state, there is no path leading to a situation in which the number of infected agents exceeds a given maximum threshold. Our basic observable is the infection state of an agent denoted by, say, a Boolean attribute $infected : \{true, false\}$. Since we are not interested in each agent's individual state but rather in the overall state of the population, we perform

[1] We view spatial relations as a particular type of arithmetic relations since they can be formulated as relations over variables describing the position of an agent.

numerical aggregation by means of counting all agents which are infected, expressed as an aggregation attribute '$numInfected$'. We could now describe our stability requirements as the number of infected agents staying below a given threshold t, i.e. '$numInfected < t$'. This expression represents a simple event. The simple event is formulated as an invariant and should therefore hold in all states of the simulation. Persistence over time describes a complex event which can be formulated semi-formally as '$always(numInfected < t)$'. In order to be able to express temporal relationships, a property specification language needs to provide support for *tense* in the form of temporal operators such as $always$, or $eventually$.

Requirement 4. In order to support the formulation of complex events, a correctness criterion language needs to provide support for tense, i.e. temporal relations between simple events.

In addition to spatial and temporal events, Page also mentions *emergent distributions* as another type of emergent macro-level behaviour that is not built explicitly into the behavioural rules of an individual agent [195]. We believe that emergent distributions can also be encoded into the notions of simple and complex events. A particular distribution of values at time t is a simple event. The evolution of the distribution over time is a complex event.

4.3 Property types

Given the notion of events, we can now consider different types of properties that can be formulated over individual runs. There are different dimensions according to which properties can be classified. First, according to their temporal nature, they can be classified into *reachability*, *safety* and *liveness properties* — a distinction commonly found in the verification literature. A reachability property (which is also often referred to as a *co-safety property*) states that a certain state or subset of states will eventually be reached. Any satisfied reachability property has a finite 'good' prefix: it can be satisfied on finite but violated on infinite paths only. A safety property states that *something bad never happens*, i.e. that an erroneous or undesired situation never occurs. Safety properties can be considered invariants in general[2]. Any violated safety properties has a finite 'bad' prefix: it can be violated on finite but satisfied on infinite paths only. A liveness property states that *something good always happens*, i.e. that a certain state (in which, e.g. a request is eventually answered) can always be reached — no matter what state the

[2]Note that this is not always true; there are examples of safety properties which are not invariants. Consider, for example, the requirement that a cash dispenser will always only dispense money if the correct PIN has been entered before; this property is not a state property and therefore also not an invariant.

system is currently in. Those properties are also often described as *repeated reachability* or *progress properties*. In general, liveness properties can be satisfied or violated on infinite paths only. This discussion indicates that the presence of finite paths may pose problems for the evaluation of temporal properties which is, in fact, true. A way to solve these problems using temporal logic by adjusting the semantics accordingly is described in Section 5.2.2.

The formulation of reachability, safety and liveness properties is possible if the underlying language possesses support for *tense*, as stipulated by Requirement 4.

According to their observational levels, correctness criteria can also be grouped into *micro*, *meso* and *macro properties*. Micro properties represent statements about the behaviour of individual agents. Meso properties represent statements about groups of interacting agents and macro properties represent statements about the population as a whole. From a technical perspective, there is no real difference between meso and macro properties since both describe events with respect to collections or groups of agents. We thus only distinguish between *individual* and *group properties*. In addition to individual and group properties, we introduce *multi-level properties* and, finally, distinguish between *internal* and *external* properties, i.e. properties that can be answered only by examining the simulation output and those that need external reference data. All property types are briefly described below.

Individual properties are properties which are formulated over an individual agent's trace. A typical example could be the assertion that a particular attribute should finally have a certain value. In a more complex scenario, one might be interested in comparing a single agent's trace with historical data in order to ascertain that the simulation reproduces historical patterns with a sufficient level of accuracy (see further below). An individual property may itself be formulated as a reachability, safety or liveness property.

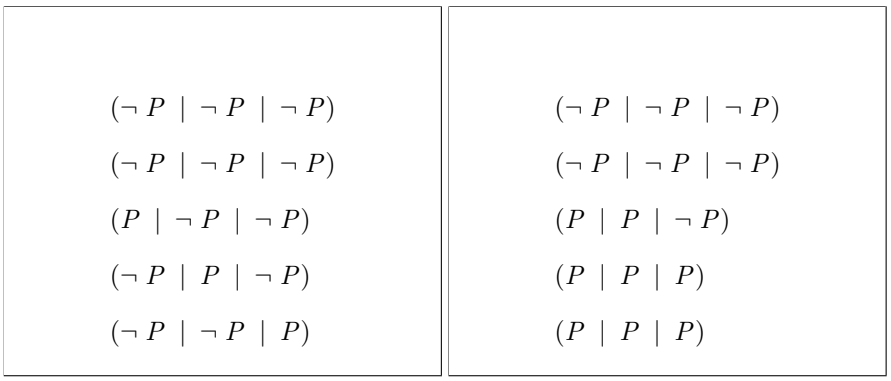
Group properties are similar to individual properties except that they make statements about the individuals within a particular group (for example the group of all male agents) or the group as a whole. They can be subdivided into *existentially*, *universally* and *numerically quantified* properties. An example of an existentially quantified property is the statement that '*there exists at least one agent that eventually knows about the new product*'. An example of a universally quantified property is the statement that '*eventually all agents will know about the new product*'. In many cases, particularly when dealing with large-scale populations, existentially and universally quantified properties are not sufficient for the expression of relevant questions. Instead, it is often necessary to explicitly quantify the amount of agents

for which some property is to hold. For example, one may want to express the fact that '*eventually at least 50% of all agents will know the new product*'.

Group properties can further be *aggregate* in nature. Whereas non-aggregate group properties (such as the ones above) view a group as a collection of individuals, aggregate ones meld individuals into a single '*meta agent*' with aggregate attributes and behaviour. For example, one may be interested in the '*total sales of the new product for all agents*' or the '*average income of all agents*'. There are situations, however, where the distinction between non-aggregate and aggregate group properties can be more difficult. Consider again the example given above: '*eventually all agents will know the new product*'. This statement can be understood in two different ways:

1. Each agent will eventually reach a state in which it knows about the new product
   ($\Rightarrow$ non-aggregate universally quantified group property)
2. A state in which every agent knows about the new product will eventually be reached
   ($\Rightarrow$ aggregate universally quantified group property)

The difference is subtle but important. The first statement makes an assumption about each agent *individually* and *separately*. The second statement is stronger; it makes an assumption *about the group of all agents*. The difference is best illustrated with an example. Consider three agents which know about a product (denoted by predicate P) or do not know about a product (denoted by the negation $\neg\ P$). Further consider the following two traces which describe the temporal evolution of the three agents' cognitive states over five time steps:



The diagram can be understood as follows. Each box represents a single possible evolution of a population of three agents for five time steps. Each row denotes the population state at a particular point

in time. For example, $(\neg\ P\ \mid\ \neg\ P\ \mid\ \neg\ P)$ describes a situation in which no agent knows about the product and $(P\ \mid\ \neg\ P\ \mid\ \neg\ P)$ describes a situation in which only agent 1 knows about the product.

The evolution shown in the left box satisfies statement 1 but not statement 2: each agent will finally know about the product *individually* (agent 1 at time 3, agent 2 at time 4 and agent 3 at time 5), but there is no global state in which *all* know about the product *simultaneously*. The evolution shown in the right box satisfies both statements: at time 4, all agents will know about the product individually *and* simultaneously. The semantic difference between the two statements clearly needs to be reflected in a property specification language.

Requirement 5. A language for the formulation of correctness criteria needs to provide support for the formulation of properties for both the *individual* and the *group level*. For group properties, *existentially*, *universally* and *numerically quantified* statements need to be supported. Furthermore, it must be possible to distinguish between *non-aggregate* and *aggregate* group properties, i.e. properties which represent statements about the aggregate behaviour of groups of agents.

The previously described property types deal with individual agents or groups of agents, respectively. However, macro, meso and micro properties need not be mutually exclusive. It is easy to imagine scenarios in which criteria are formulated both over individuals and over groups of agents or even the entire population. This leads to the idea of *multi-level properties*. Instead of being associated with a particular observational level, they are meant to describe *social dynamics* within an agent population, e.g. the influence that an agent has on its neighbours. Agent-based simulation is particularly powerful for the simulation of complex systems in which individuals communicate with and react to their neighbourhood and, in doing so, produce aggregate and emergent behaviour on the macro level. In order to analyse relationships between different observational levels, micro level analysis alone is not sufficient. On the other hand, restricting the focus of analysis to the meso or macro level may allow for the *detection* of certain higher-level phenomena, but it is just as insufficient for their *explanation* as pure micro level analysis. To this end, one may thus want to formulate more complex, multi-level properties, i.e. properties about the impact of individual agents/groups of agents on other individual agents/groups of agents. A multi-level property can transcend the borders of observational levels and can, for example, describe the diffusion of entities such as information, knowledge, products, etc. from individuals to groups to the entire population (or vice versa). A typical example is the following statement: '*If agent A knows about a product, then, eventually, all other agents who know agent A will also know about the product*'. Note that, despite its causal semblance, the statement about the occurrence of the two events (Agent A knowing about the product and all of its peers also eventually knowing about the product)

is purely temporal. The analysis of a single run cannot derive causality between two events from the fact that they just happened to occur in a particular temporal order. We return to this issue in Chapter 7 where we relate simulation runs with *possible worlds* and show that the analysis of temporal properties becomes particularly powerful if it is performed on multiple paths. In this case, correlation and causality can in fact be distinguished.

Requirement 6. A language for the formulation of correctness criteria needs to support *multi-level properties*, i.e. properties which combine statements about individual agents and groups of agents.

4.4 Including external data

The types of properties described above were *internal* in nature, i.e. they can be answered solely by analysing the information provided by a fragment of an individual simulation run. These types of question correspond with the idea of *internal validation* — the assessment of the *model-theory link*, i.e. the internal mechanisms of the model — described in Section 2.2.3. For the purpose of *external validation*, on the other hand, it is generally necessary to compare simulation output with existing data (e.g. historical sales records) and calculate the *level of correspondence* using statistical metrics (e.g. the *squared error*). As described in Section 2.2.3, external validation focusses on the *model-phenomenon link*, i.e. the model's representativeness of the phenomenon in the real world. External validation is a large area in its own right and largely focussed on statistical techniques [139, 177]. The focus of this work is on internal validation. Nevertheless, as described in Section 2.2.3, sometimes the boundary between internal and external verification is blurred and it is, for example, useful to compare the occurrence of certain phenomena in the simulation with an external reference data set. The purpose of this section is to give an example of how the technique described in this work can be used to answer basic questions of this type.

Consider, for example, a consumer purchase simulation which produces as its output aggregate sales data on the macro level (see Figure 4.1). We assume that historical sales data is also available as a separate time series. In this case, the purpose of external validation is to analyse the correspondence between the artificial model and the measured real-world data.

Similar to internal validation or verification, external validation can also take place on different observational levels. External validity on the micro level refers to the conformity of an individual agent's behaviour with a given reference model. For example, one could compare the purchasing behaviour of

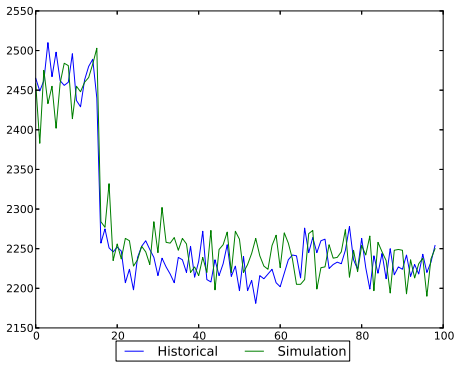


FIGURE 4.1: Time series of historical and simulated sales levels

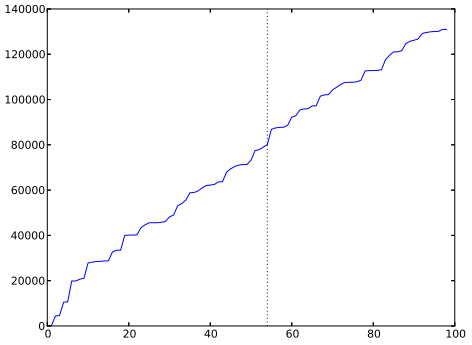


FIGURE 4.2: Cumulative squared error (CSE) and correctness threshold

an artificial agent with historical data about individual sales that have been obtained through qualitative and quantitative research. External validity on the meso level represents the conformity of a group of agents with their representatives in the reference model. A typical approach could, for example, be to segment the population into male and female agents or into different age bands. On the macro level, an agent-based simulation typically produces a sequence of aggregate values which is then assessed against the background of existing data. An example of the comparison of simulated and historical aggregate data has been given above. We now show how time series comparison can be transformed into a safety checking problem. Suppose we have two time series $X = \langle x_1, x_2, ..., x_n \rangle$ and $Y = \langle y_1, y_2, ..., y_n \rangle$ and a metric $M : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ which allows us to quantify the difference of two data points x_i and y_i (e.g. the squared error). Then we can construct a third time series $Z = \langle z_i \mid i \in [1, n] \wedge z = \sum_{j=0}^{i} M(x_j, y_j) \rangle$ where each z_i represents a *distance measure* between two states x_i and y_i, for example the *Cumulative Squared Error (CSE)*. Given this time series, we can now check the correctness of the property by means of safety checking. We first need to translate time series Z into a state transition system in which each state represents one data point $z_i \in Z$ and is also labelled accordingly. The criterion that the error between X and Y should never exceed a given threshold ϵ can now be stated semi-formally as a safety property as '$always(z \leq \epsilon)$'.

Consider again the consumer purchase example which produces aggregate sales data. Given the two time series (simulation and historical) shown in Figure 4.1, we can easily construct a third time series Z according to the description given above. An example time series using CSE is shown in Figure 4.2. We might now expect the error between the two time series to stay below 8,000 (depicted by the dotted

vertical line) in order for the simulation output to be accurate. This can be easily shown as described above.

Requirement 7. A language for the formulation of correctness criteria needs to support the *integration of external data* for the purpose of measuring the representational accuracy of the simulation output with respect to historical data.

4.5 Summary

In this chapter, we gathered *requirements* for the design of a formal language for describing correctness criteria of agent-based simulations. Starting with *atomic* and *composite observables* (the former of which refer to individual attributes and the latter of which to aggregations thereof), the notion of *events* was then discussed in further detail. We distinguished between *simple* and *complex events* and described them as Boolean functions mapping from group states to either $true$ or $false$. We further described different *types of properties* that are commonly found in the analysis of agent-based simulations against the background of two dimensions, *temporal nature* and *observational level*. In the case of group properties, we further need to distinguish between *existentially*, *universally* and *numerically quantified* as well as between *non-aggregate* and *aggregate properties*. We also found that observational levels are not mutually exclusive within single properties; *multi-level properties* which describe, for example, the diffusion of information, knowledge, ownership of products, etc., within the population, need also be expressible. And finally, we concluded that, in order to be able to answer certain types of questions at the boundary of internal and external validation, it must be possible to integrate *external reference data* into the analysis process in order to compare the simulation output with it.

So far, the description has been informal and the properties were given in natural language. In order for them to be satisfied in an automated and rigorous way, however, a formalisation is necessary. Due to the complex nature of agent-based simulation, an appropriate formal language needs to provide a high level of granularity in order to satisfy the requirements gathered in this chapter. The next chapter introduces *simLTL*, a temporal logic-based specification language tailored to the formulation of simulation-related correctness properties over individual simulation runs.

Chapter 5

simLTL: A LTL-based property specification language

5.1 Introduction

Designing good specification languages has always been a central research topic in the area of runtime verification. In the previous chapter, various requirements that a property specification language for agent-based simulations needs to satisfy have been identified and described informally. This chapter describes the design of simLTL, a novel LTL-based specification language which incorporates these requirements. We start with the description of a simple yet illustrative agent-based simulation model in Section 5.1.1 which serves as the running example in this work, followed by a brief overview of existing specification languages in Section 5.1.2. We then provide an overview of design decisions, the semantic model, and issues related with the evaluation of simLTL properties on finite traces in Section 5.2. The agent layer, i.e. the temporal core of simLTL, including its syntax and semantics with respect to individual agent traces, is described in Section 5.3. We then extend our focus to the global layer and describe both syntax and semantics of full simLTL formulae which are formulated over simulation traces in Section 5.4. The analysis of agent-based simulations often involves complex calculations, e.g. in order to determine the correlation between the simulation outcome and a historic data set (as described in Section 4.4). Instead of integrating, for example, statistical functions as first-class citizens into the property specification language (which would lead to an unnecessary blow-up), we instead

assume their presence in the underlying simulation and refer to them with placeholders in the form of attributes, aggregations, and predicates. This idea is described in Section 5.5 in further detail. The usage of simLTL for the formulation of typical agent-based correctness criteria is illustrated with a range of example properties in Section 5.6. The chapter concludes with a summary in Section 5.7.

5.1.1 Motivating example

Algorithm 2 Outline of the agent update function

```
p := number of infected neighbours
n := total number of neighbours
if state = Susceptible then
    move to state Infected with probability p/n
    remain in state Susceptible with probability 1 − (p/n)
else {state = Infected}
    move to state Recovered with probability 0.3
    remain in state Infected with probability 0.7
else {state = Recovered}
    remain in state Recovered with probability 1.0
end if
```

For the purpose of illustration, we revisit and extend here our transmission model introduced in Section 1.2.1 and further specified in Section 3.4.5. In the model, each agent can be in one of three possible states, 'S' ($susceptible$), 'I' ($infected$), or 'R' ($recovered$). We further assume that state transitions are probabilistic and solely dependent upon the infection state of an agent's neighbourhood. An example protocol that a single agent may follow is shown in Algorithm 2. In certain variants of the model, the agent may also be allowed to transition from R back to S. For certain example properties, we assume the existence of additional agent attributes (e.g. $gender$). Further explanations are given as part of the description of the respective property.

Let us now turn to the types of correctness criteria that one may want to formulate over such a transmission model. Starting on the individual level, the first check concerns the *correctness of individual state transitions*. We want to make sure that all agents adhere to the following state transition scheme: $S \rightarrow I \rightarrow R(\rightarrow S)$. We further want to allow each agent to stay in its current state indefinitely.

The second type of property describes the *correct triggering of individual transitions*. For example, we want to make sure that a susceptible agent becomes infected only if a certain fraction of its neighbourhood, say 80%, is also infected.

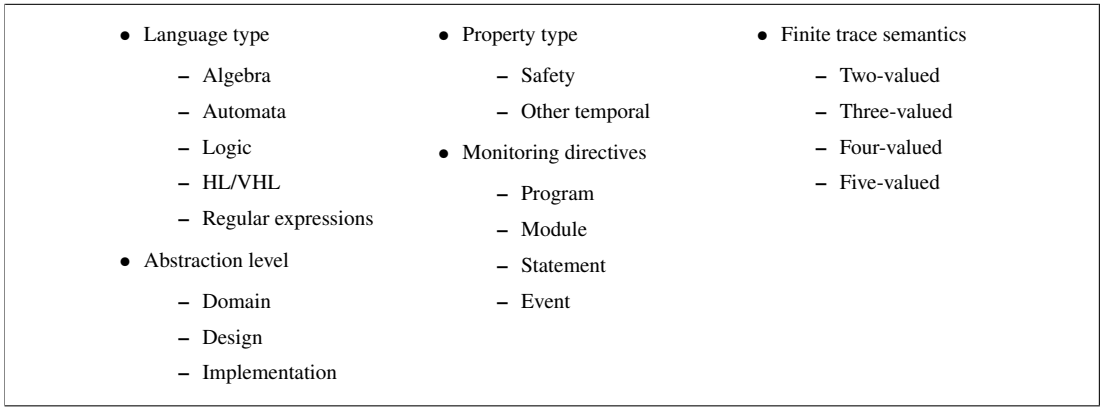


FIGURE 5.1: A taxonomy for runtime verification languages (adapted from [72])

The third property type concerns the *correctness of the temporal aggregate behaviour of the population*. This involves, for example, criteria which state that the population as a whole will never reach an invalid state or that a certain aggregate value will always remain within given bounds.

In the next section, we provide an overview of existing specification languages for runtime verification.

5.1.2 Specification languages for runtime verification

According to the taxonomy given by Delgado *et al.* [73] (which we slightly extended), existing specification languages can be classified according to five major criteria, as shown in Figure 5.1. The *language type* describes the actual mechanism used to describe the property that needs to be satisfied. Algebra-based approaches use pre- and postconditions and thus follow the idea of *design-by-contract* [184]. An example of this category is *Java with assertions (Jass)*, a preprocessor which translates annotated Java programs into pure Java programs where the checking logic is already embedded [21]. Automata-based approaches transform a temporal logic formula into an automata representation which represents all allowed progressions of the system under study and acts as a monitor. An example of this category is the RuleR system [20]. Logic-based approaches operate directly upon the underlying formula which is mostly formulated in a variant of LTL. Examples in this category are the *Meta-Event Definition Language (MEDL)*, a LTL-based requirements specification language used in the *Monitoring and Checking (MaC)* framework [151] as well as the *Property Specification Language (PSL)* [83]. Other approaches allow for the formulation of properties in a high-level (HL) or even very high-level (VHL) programming language — often the language that the system under study is written in. An example of this category is

BEE++ which provides an application framework where requirements are represented by *events* which are themselves formulated in C++ and assembled from lower-level components in an object-oriented fashion through inheritance [40]. And finally, some approaches also allow for the specification of system properties as *regular expressions*. An example of this category is *tracematches* [2].

The *abstraction level* describes the level upon which verification properties can be formulated. Domain-specific approaches allow for the formulation of a wide range of properties related to a particular domain — e.g. interaction in multiagent system [13] — often irrespective of the underlying programming language. Design-based approaches are targeted towards particular architectural constructs within the system under study, e.g. pre- or postconditions of program methods. Implementation-specific approaches are most specific and allow for the specification of programming language-dependent criteria, e.g. memory safety in C programs [209].

In terms of *property types*, Delgado *et al.* distinguish between *safety* and *other temporal properties*. As described in Section 4.3, safety properties are widely used in verification; they state that 'something bad will never happen' and act as *invariants* which are always required to hold. Other important types of temporal properties such as *liveness*, *fairness* or *timing properties* are often essentially different from safety properties and not expressible as invariants.

The *monitoring directives* refer to the level of granularity with respect to property formulation. Program-level languages allow for the formulation of properties that relate to, for example, the full set of states or to global data structures. Module-level languages allow for the formulation of properties about program modules such as classes, abstract data types or program methods; typical examples are the aforementioned pre- and postconditions. Event-based properties are formulated about 'things that happen', i.e. changes in state. For example, it may be necessary to ensure the correct assignment of values to a data structure at runtime. In this case, the check needs to be invoked every time an assignment takes place.

Specification languages can be further classified according to their *semantics* with respect to finite traces. As briefly mentioned in Section 2.4.2, conventional LTL assumes the existence of infinite traces. In contrast, traces obtained from real system executions are necessarily finite. A property specification language needs to take finiteness into account by adjusting the semantics accordingly. A wide range of semantics for LTL in the presence of finite traces have been presented in literature [24]. We can generally distinguish between *two-valued*, *three-valued*, *four-valued* and *five-valued* logics. In the two-valued scenario, properties can be either true or false [173]. In the three-valued case, properties may be *true*, *false*, or *undecidable/unknown* [23]. In the four-valued case, properties may be *true*, *false*, *possibly*

true, or *possibly false*[1] [24]. In the five-valued case, properties may be *true*, *false*, *possibly true*, *possibly false*, or *undecidable* [190]. Issues with finite traces are described in more detail in Section 5.2.2 below.

5.2 Design decisions

Most existing languages have in common that they allow for the verification of temporal properties against the background of a wide range of systems at runtime. It is thus natural to consider their application to the verification of agent-based simulations. We focus on two particular characteristics of agent-based simulations below which, we argue, existing languages are not sufficiently able to deal with.

The first particular characteristic of agent-based simulations is their heterogeneous nature. Rather than a mere set of attributes, each agent has its own *life history* which may be subject to individual correctness checking. As described in the previous chapter, rather than just about the system as a whole, we also need to be able to formulate and verify statements about individual agents. Furthermore, in the presence of a large number of agents, we need to be able to *quantify* efficiently. Instead of enumerating individual agents explicitly within a property, we want to formulate statements like, for example, 'all agents satisfy property p', 'at least 10 agents satisfy property p' or 'at most 80% of the agents satisfy p'. None of the languages above provides such a quantification mechanism 'out-of-the-box', without further extensions.

The second characteristic is that agent-based simulations are *multi-layered*. Each simulation can be seen as a set of individual agents or as a set of (possibly overlapping) groups of agents — the number of which can be large. For example, given a set A of n agents, each member of $\mathbb{F}\, A$ (the set of finite subsets of A) is one possible group of agents that correctness criteria may need to be formulated upon. Each group can itself either be seen as a homogeneous entity — a 'meta-agent' — or as a collection of individuals. Whichever view is taken, it opens up a different dimension for correctness checking. For a specification language, this means that we need to be able to restrict properties to certain *observational levels* by means of *selecting* groups of agents based on certain commonalities, e.g. common attribute values. For example, we may want to restrict the focus of a property to the group of male or female agents only. The only language which provides an appropriate mechanism is $\mathcal{ALC}$-LTL (a hybrid language which combines temporal logic and description logic) by means of *roles*, *concepts* and *individuals* [11]. This,

[1] Names may vary from approach to approach.

however, comes at a cost since a mapping between the trace being verified and the roles, concepts, and individuals needs to be established prior to verification.

As described in Sections 2.1.1 and 3.2, simulation-based agents are typically much simpler than their non-simulation counterparts. As opposed to the area of multiagent systems where cognitive architectures such as BDI, Soar, or ACT-R have been used successfully to construct behaviourally rich, deliberative agents, those in a simulation context are, in most cases, implemented in a very simple, rule-based way. The temporal behaviour of individuals is rarely goal-driven and reduces to simple changes in the agent's belief base — most often a simple key-value store. The primary reason for that is that the focus of interest in agent-based simulation is typically more on the *social dynamics* of the simulated population than on the internal decision-making process of individual agents. As a consequence, the requirements for verification in a simulation context differ from that in the more general area of multiagent systems (see Section 2.5). Although there is an increasing interest in more behavioural sophistication in agent-based simulation [106, 36, 42], the simple, rule-based approach can be expected to remain prevalent; we thus restrict the focus of verification to the purely temporal behaviour of agent population and leave BDI concepts and epistemic concepts as future work (see Section 10.3).

In the following sections, we describe the design of simLTL, a novel LTL-based specification language which aims to satisfy the requirements identified in the previous chapter. With respect to the taxonomy shown in Figure 5.1, simLTL is (i) *logic-based*, (ii) located at the *domain level* (tailored to agent-based simulations), (iii) designed to express *different types of temporal criteria* (not just safety properties), (iii) *program-based*, and (iv) *two-valued*, i.e. each property is either true or false.

As briefly mentioned in Section 2.4.2, temporal logics can be generally subdivided into two major classes: *linear temporal logic* (LTL) and *branching time logic* (CTL,CTL*). In the former case, time is seen as a linear sequence of time steps; in the latter case, each state may provide a choice of possible successor states. It is obvious that, depending on which model of time is used, different properties become formulable. Branching time properties are *state properties*, i.e. their semantics are defined over states; linear time properties are *path properties*, i.e. their semantics are defined over entire paths. Due to the possibility of multiple successor states, branching time logics allow for *nested quantified statements*. For example, in CTL, it is possible to formulate the following statement which is not expressible in LTL: *"for all possible executions, it will always be true that there exists an execution in which ϕ is eventually true"* ($\forall \mathbf{G} \exists \mathbf{F} \phi$). On the other hand, the following statement can be expressed in LTL but not in CTL: *"It will be eventually the case that ϕ holds forever"* ($\mathbf{FG}\phi$). LTL and CTL thus have different expressive power. CTL*, on the other hand, is strictly more expressive. It subsumes both LTL and CTL and allows

for the formulation of properties that are neither expressive in LTL nor in CTL. An example is the statement that *"for all executions, it will eventually be the case that ϕ holds forever"* ($\forall \mathbf{FG}\phi$ [12]).

The first decision that had to be made with respect to a logic-based verification language was whether it assumes a linear or a branching flow of time. As described above, the latter can be strictly more expressive than the former. One design decision of the approach was that it is allows for both *offline* and *online verification*. In the former case, correctness properties are checked *a posteriori*, i.e. on simulation traces that have been produced independently from the verification process; in the latter case, properties are checked *on-the-fly*, i.e. while the simulation is running. Whereas in this case, it is theoretically possible to 'steer' the simulation such that branching time properties become answerable (see Section 10.3.1), we limit our focus to a linear time setting in this work. Answering branching time properties complicates the underlying simulation process significantly. It requires an exhaustive traversal of the state space which, in a probabilistic setting, can be realised by repeatedly sampling successor states; this, however, is generally not supported by existing simulation environments. We aim to explore the possibility to answer branching time properties as part of our future work.

Within the universe of linear temporal logic, we further need to clarify the model of time that we operate on. Conventional LTL is based on a semantic model that emphasises *observations* (i.e. values of propositions) and their *relative order* but not their *precise timing* [12]. This has important implications on the verification of real-time properties, i.e. properties that make statements about the real-time behaviour of the system under consideration. In conventional LTL, this is not easily possible. Since there is no notion of precise time points at which events happen, it is impossible to formulate statements such as *"ϕ must happen within at most 5 time units"*. In order to solve this problem, several timed version of LTL have been proposed, for example *Metric Temporal Logic (MTL)* [145], *Timed Propositional LTL (TPLTL)* [3], or *Timed LTL (TLTL)* [204]. In MTL, modalities are augmented with *timing constraints*; this allows for the expression of the statement above as $\mathbf{F}_{(0,5)}\phi$. In TPLTL, time points are represented explicitly as first-order variables ranging over the time domain; the statement above can then be expressed as $\mathbf{F}x.(\phi \wedge x \leq 5)$. TLTL incorporates modalities of the form $\lhd_a \in I$ and $\rhd_a \in I$, respectively, into the language. They state that the time elapsed since the last occurrence of a (the time until the next occurrence of a, respectively) lies within the time interval I. The statement above can then be formulated as $\rhd_\phi \in [0, 5]$.

As described in Section 3.3, the state of the population is typically updated in discrete and uniform time steps. Depending on the problem to be simulated, the granularity of time steps can be either very small (e.g. seconds) or very large (e.g. months, years, etc.). At the end of each time step, the simulation is

assumed to have updated the entire population of agents. Even though a discrete event in which events occur at arbitrary points in the continuous space of time may be equally appropriate in the context of agent-based simulation, the complementary, discrete time approach is typically more prevalent. As a consequence, we chose to focus on a discrete notion of time found in conventional LTL and leave the verification of real-time properties as a possible task for future work.

In order to provide the required level of granularity, simLTL comprises two different layers, the *agent* and the *global layer*. Agent formulae (i.e. those formulated within the agent layer) are interpreted on agent traces and support the formulation of individual temporal properties. Full simLTL formulae are interpreted on simulation traces and support the formulation of group-based and aggregate properties. As described further below, this subdivision facilitates the realisation of ideas such as *selection* and *quantification*.

It is important to emphasise that simLTL is designed as a property specification language rather than as a full temporal logic used for proof-theoretic purposes. As a consequence, we shall not discuss soundness and completeness in this work. Before describing both syntax and semantics of simLTL, it is helpful to briefly revisit the semantic model, i.e. the formal representation of the types of simulations that we aim to formulate properties about, in the next section.

5.2.1 Semantic model

A formal description of agent-based simulations was presented in Chapter 3 and slightly extended in Chapter 4. In this section, we briefly revisit the central points in order to clarify the semantic model that simLTL formulae are to be evaluated upon. The focus of this work is on the temporal evolution of an agent-based model and we thus view an agent's evolution over time as a simple sequence of states. We refer to such a sequence as an *agent trace*:

$$Tr_a == \text{seq}\, AState$$

where $AState == Name \rightarrow Value$ is a mapping from the set of attribute names to the set of attribute values. Each group of agents is assumed to be finite and composed of n individual agents. Consequently, a *group state* is defined as a function from the set of agent identifiers Ag to the set of agent states $AState$:

$$GState == Ag \rightarrow AState$$

Since agents evolve over time, groups are also evolving. A *simulation trace* is thus defined as a sequence of group states:

$$Tr_s == \text{seq } GState$$

In order to provide access to aggregate attributes on both the individual and the group level, we assume the existence of an external *aggregation function* on both the agent and the group level which is represented formally as a function from the set of attribute names and the set of agent or group states to the set of attribute values (see Section 4.2.1 for details):

$$A_a == Name \times AState \rightarrow Value$$
$$A_g == Name \times GState \rightarrow Value$$

We further assume the existence of *agent* and *group predicate functions* as described in Section 4.2.2. Formally, an agent predicate function P_a is a n-ary function which accepts a predicate name, an agent state and a finite set of arguments and returns a Boolean value. A group predicate function can be defined accordingly:

$$P_a : Name \times AState \times \mathbb{F}\ Value \rightarrow \{\text{true}, \text{false}\}$$
$$P_g : Name \times GState \times \mathbb{F}\ Value \rightarrow \{\text{true}, \text{false}\}$$

Let $t_s : Tr_s$ be a simulation trace. We then denote its size with $\#t_s$, and refer to the group state at time t as $t_s[t]$. If we want to refer to the state of agent a at time t, we write $t_s[t][a]$. Consequently, if we want to refer to the atomic, non-aggregate attribute x of agent a at time t we write $t_s[t][a][x]$. Aggregate attributes can be accessed via the aggregation function. For example, in order to access aggregate attribute y of agent a at time t, we write $A_a(y, t_s[t][a])$.

It is also often necessary to refer to the individual agent traces within a simulation trace. Since a simulation trace is defined as a set of group states rather than as a set of agent traces, the set containment operator cannot be used straight away. In order to facilitate access, we first define ids as another convenience function which returns the set of all agent identifiers in a given group state:

$$ids : GState \rightarrow \mathbb{F}\ Value$$
$$\forall\ gs : GState \bullet ids(gs) = \text{dom } gs$$

We further define a mapping function $traces_a$ which maps from a simulation trace to the set of contained agent traces as follows:

$$traces_a : Tr_s \rightarrow \mathbb{F}\ Tr_a$$
$$\forall\, t_s : Tr_s \bullet filter(t_s) = \langle ags \lhd t_s[t] \mid t \in 1\,..\,\#t_s \rangle$$

The final convenience function introduced is *filter*. It accepts a simulation trace as well as a set of agent ids as arguments and returns a simulation trace comprising only those agents with the given ids.

$$filter : Tr_s \times \mathbb{F}\ Ag \rightarrow Tr_s$$
$$\forall\, t_s : Tr_s \bullet (\forall\, ags : \mathbb{F}\ Ag \bullet filter(t_s, ids) = \langle \langle glh[t] \mid glh[t].1 \in ags \rangle \mid t \in 1..\#t_s \rangle$$

5.2.2 The finite trace problem

An important characteristic of the idea of traces obtained through simulation is that they are inherently time-bounded. However, as described in Section 2.4.2, LTL is defined over infinite traces. This results in a problem which needs to be dealt with in the semantics of the logic being used.

In order to illustrate this problem, it is important to introduce the idea of *expansion laws*. Informally, expansion laws allow for the decomposition of a LTL formulae into two parts: the fragment of the formula that needs to hold in the *current* state and the fragment that needs to hold in the *next* state in order for the whole formula to be true. It is useful to view both fragments as *obligations*, i.e. aspects of the formula that the trace under consideration needs to satisfy immediately and aspects that it promises to satisfy in the next step. In the following sections, we thus refer to both fragments as *immediate* and *future obligations*, respectively. The expansion laws that hold for conventional LTL are shown below:

$$\phi_1\ \mathbf{U}\ \phi_2 \equiv \phi_2 \vee (\phi_1 \wedge \mathbf{X}(\phi_1\ \mathbf{U}\ \phi_2)) \tag{5.1}$$
$$\phi_1\ \mathbf{R}\ \phi_2 \equiv \phi_2 \wedge (\phi_1 \vee \mathbf{X}(\phi_1\ \mathbf{R}\ \phi_2)) \tag{5.2}$$

Expansion laws play an important role for the idea of runtime verification, since they form the basis for a decision procedure which a model checker can use to decide in a certain state whether a given property has already been satisfied or violated. By decomposing a formula into immediate and future obligations, optimality can be achieved: as soon as the immediate obligation is satisfied and no future obligation has been created, the entire formula is satisfied and the evaluation finishes.

Expansion laws work well if the trace under consideration is infinite. In the presence of finite traces, however, certain problems arise. It is unclear how the semantics of 'next' should be defined for the

final state of a trace. Under conventional interpretation, in order for $\mathbf{X}\phi$ to be true, the existence of a successor state is strictly required (this interpretation is subsequently being referred to as *existential next*); any evaluation of the formula will thus always produce an obligation. In the absence of a successor state, however, the obligation can never be satisfied which makes the evaluation of 'next' generally undecidable in the final state of a finite trace.

Since 'next' forms the basis for the expansion laws for 'until' and 'release', the problem propagates. Consider, for example, the evaluation of the expanded 'until' formula as defined by expansion law 5.1 above in the final state of a trace. The formula is a disjunction of its first subformula with a 'next' statement as its second operand. Since the 'next' statement is undecidable in the final state, the whole formula is only decidable if the evaluation of its second operand can be avoided. This is only the case if its first operand evaluates to true. If we are in the final state of a trace and the first operand evaluates to false, however, the whole formula becomes undecidable. This leads to the following problem:

Problem 1. Under conventional semantics for 'next', when being evaluated on a finite trace, an 'until' formula can only either be satisfied or undecidable, but never refuted.

The problem becomes even more apparent when taking into account additional temporal operators. In conventional LTL, 'globally' and 'finally' are defined in terms of 'until' as follows:

$$\mathbf{F}\phi \equiv \text{true } \mathbf{U}\ \phi \tag{5.3}$$

$$\mathbf{G}\phi \equiv \neg\, \mathbf{F} \neg\, \phi \tag{5.4}$$

In the case of infinite traces, these equivalences are perfectly valid, $\mathbf{G}\phi$ can be taken literally: ϕ is expected to hold *now and forever* in the future. In a finite world, however, the aforementioned issues of undecidability arise. Since both operators are based on the 'until' operator, we get the following problem:

Problem 2. Under conventional semantics for 'next', when being evaluated on a finite trace, a 'finally' formula can only either be satisfied or undecidable, but never refuted. Conversely, a 'globally' formula can only either be refuted or undecidable, but never satisfied.

This clearly clashes with the intuitive meaning of 'finally' and 'globally' in our case. In order to understand why, it is important to clarify the difference between *truncated* and *maximal* (or *complete*) traces. A truncated trace can be seen as a *finite prefix* of an otherwise infinite trace. A typical reason for working with truncated traces is to circumvent state space explosion, e.g. by using *Bounded Model*

Checking [31]. Since a truncated trace is a finite prefix of a (possibly infinite) number of continuations, however, the trace may not be sufficient for determining the truth or falsity of a property (as illustrated above). A maximal trace, on the other hand, can be assumed to be complete (or *terminated*). It does not represent a finite prefix of an infinite path and, therefore, there is no unknown future continuation which needs to be taken into account when evaluating a property.

Undecidability of a property makes sense in the case of a truncated trace; if the finite prefix currently being examined is simply too short to make a conclusive decision, then it is reasonable to end the evaluation without a conclusive result. In the case of maximal paths, however, undecidability is not an option. Since maximal traces are complete, referring to a (non-existent) point in the future at which the property may be satisfied or refuted does certainly not make sense. In the maximal case, we expect $\mathbf{F}\phi$ to be true precisely if ϕ is true *in some state along the trace*; consequently, we expect $\mathbf{G}\phi$ to be true precisely if ϕ is true *in all states along the trace*.

The finite trace problem has been discussed extensively in literature, a good overview has been given by Bauer *et al.* [24]. Two general solution approaches can be distinguished. The first approach which has, for example, been followed by Manna and Pnueli [173] and by Eisner *et al.* [84] is to keep the two-valued semantics of conventional LTL and extend them to finite traces; the second approach which has, for example, been followed by Bauer *et al.* [24] and Morgenstern *et al.* [192] is to extend the domain of truth values to three (*true*, *false*, and *inconclusive*), to four (*certainly true*, *certainly false*, *presumably true*, and *presumably false*) or even to five values (*certainly true*, *certainly false*, *possibly true*, *possibly false*, and *inconclusive*) [190].

A typical property of individual runs of an agent-based simulation is that they are time-bounded. This poses an upper limit on the number of states within any simulation trace produced by a simulation; traces obtained by agent-based simulations can thus be seen as being maximal rather than as being truncated. It has been shown that Manna and Pnueli's approach possesses semantics which are suitable for maximal traces [24]. We therefore build upon their ideas in this work and incorporate the semantics of FLTL — their variant of LTL — defined over finite traces into our specification language. The choice for FLTL is motivated in more detail in the following paragraphs.

As opposed to a single 'next' operator, FLTL comprises both a *weak* and a *strong* version of 'next'. On non-final states, both versions are equivalent. In the final state, however, their semantics differ. In the absence of a successor state, the weak version of 'next' (denoted $\mathbf{X}_\forall$) always evaluates to true; it is also

called *universal next*. The strong version (denoted $\mathbf{X}_\exists$), on the other hand, will always evaluate to false if there is no successor state; it is also called *existential next*.

The necessity for differentiating between a universal and an existential semantics for 'next' becomes clearer in the context of the undecidability of 'finally' and 'globally' under conventional semantics as described above: a 'finally' property becomes undecidable if the final state has been reached but the property has not been satisfied yet. This can be seen as failing to show the *existence* of a possible continuation which will ultimately render the property true. Conversely, a 'globally' property becomes undecidable if the final state has been reached but the property has not been refuted yet; this can be seen as failing to show that *all* possible continuations of the trace will also satisfy the property *universally*.

In order to reflect this difference, the expansion laws for FLTL need to be modified as follows:

$$\phi_1 \mathbf{U} \phi_2 \equiv \phi_2 \vee (\phi_1 \wedge \mathbf{X}_\exists(\phi_1 \mathbf{U} \phi_2)) \tag{5.5}$$

$$\phi_1 \mathbf{R} \phi_2 \equiv \phi_2 \wedge (\phi_1 \vee \mathbf{X}_\forall(\phi_1 \mathbf{R} \phi_2)) \tag{5.6}$$

The usefulness of these laws becomes more apparent when considering the expansion of 'finally' and 'globally'. Consider 'finally' first:

$$\begin{aligned}
\mathbf{F}\phi &\equiv \text{true } \mathbf{U} \phi \\
&\equiv \phi \vee (\text{true} \wedge \mathbf{X}_\exists(\text{true } \mathbf{U} \phi)) \\
&\equiv \phi \vee \mathbf{X}_\exists(\text{true } \mathbf{U} \phi) \\
&\equiv \phi \vee \mathbf{X}_\exists\mathbf{F}\phi
\end{aligned} \tag{5.7}$$

Under these semantics, $\mathbf{F}\phi$ evaluates to true if and only if ϕ is satisfied in *any* state along the finite trace; this corresponds with the intuitive meaning of 'finally' against the background of maximal traces mentioned above. Consider now the expansion of 'globally':

$$\begin{aligned}
\mathbf{G}\phi &\equiv \neg\, \mathbf{F} \neg\, \phi \\
&\equiv \neg\, (\text{true } \mathbf{U} \neg\, \phi) \\
&\equiv \text{false } \mathbf{R}\, \phi
\end{aligned}$$

$$\begin{aligned} &\equiv \phi \wedge (\text{false} \vee \mathbf{X}_{\forall}(\text{false } \mathbf{R}\, \phi) \\ &\equiv \phi \wedge \mathbf{X}_{\forall}(\text{false } \mathbf{R}\, \phi) \\ &\equiv \phi \wedge \mathbf{X}_{\forall}\mathbf{G}\phi \end{aligned} \tag{5.8}$$

Under these semantics, $\mathbf{G}\phi$ evaluates to true if and only if ϕ is satisfied in *all* states along the path; again, this corresponds with the intuitive meaning of 'globally' against the background of maximal traces mentioned above.

The usefulness of FLTL for our purposes can also be established more formally. In order to define requirements for different variants of LTL in the context of runtime verification, Bauer *et al.* postulate four maxims which a semantics on finite traces needs to satisfy in order to be as close to the semantics of conventional LTL over infinite paths as possible [24]:

1. *Existential next* requires the inclusion of a strong next operator
2. *Complementation by negation* requires that a negated formula evaluates to the complemented and different truth value
3. *Impartiality* requires that a finite trace is not evaluated to true (or false) if there still exists an infinite continuation leading to another verdict, and
4. *Anticipation* requires that once every infinite continuation of a finite trace leads to the same verdict, then the finite trace evaluates to this very same verdict

In order to understand Maxim 1, it is helpful to revisit a fundamental law which is satisfied by LTL in the context of infinite traces, but which may cause problems in the case of finite traces. It describes the logical equivalence of 'true' and '$\mathbf{X}$ true' (we use '$\equiv_\omega$' to denote logical equivalence in the case of infinite paths):

$$\text{true} \equiv_\omega \mathbf{X}\, \text{true} \tag{5.9}$$

Intuitively, this law states that there is always a successor state in which 'true' holds. In FLTL, under the weak interpretation of 'next', this does not hold in the final state of a trace; under the strong interpretation, however, the law holds. The existence of a strong operator is exactly what Maxim 1 requires; it is thus satisfied by FLTL.

Maxim 2 is motivated by a second law which is satisfied by LTL over infinite traces. It describes the negation of the 'next' operator:

$$\neg\, \mathbf{X}\phi \equiv_{\omega} \mathbf{X}\neg\, \phi \tag{5.10}$$

This law can be intuitively understood as follows; whenever $\neg\ \mathbf{X}\phi$ is true in a state, then ϕ must be false in its direct successor state; both LTL and FLTL satisfy this requirement. In the case of LTL, this is easy to see. For FLTL, it is not immediately obvious. In FLTL, negation of 'next' is treated differently than in LTL; equivalence 5.10 holds for LTL, but not for FLTL. Instead, FLTL satisfies the following equivalence:

$$\neg\, \mathbf{X}_{\exists}\phi \equiv \mathbf{X}_{\forall}\neg\, \phi \tag{5.11}$$

Despite its different flavour, FLTL is still defined over a two-valued truth domain where complementary values differ; as a consequence, Maxim 2 holds.

Maxims 3 and 4 concern the semantics with respect to the future behaviour of the underlying system, i.e., to possible continuations of the finite trace currently being analysed. As mentioned further above, this is only relevant for truncated, not for maximal paths; as a consequence, Maxims 3 and 4 need not concern us further. We can therefore conclude that the semantics of FLTL defined over finite traces are suitable for the runtime verification problem in the context of agent-based simulations.

Another problem which is worth mentioning is the semantics of nested temporal operators such as '$\mathbf{FG}$' or '$\mathbf{GF}$' in the case of finite traces. Both combinations are commonly used in verification. Intuitively, $\mathbf{FG}\phi$ states that ϕ will eventually always be satisfied and $\mathbf{GF}\phi$ states that there will always be a point in the future in which ϕ holds[2]. Following the laws defined above, the two nested operators expand as follows:

$$\mathbf{FG}\phi \equiv (\phi \wedge \mathbf{X}_{\forall}\phi) \vee \mathbf{X}_{\exists}\mathbf{FG}\phi \tag{5.12}$$

$$\mathbf{GF}\phi \equiv (\phi \vee \mathbf{X}_{\exists}\phi) \wedge \mathbf{X}_{\forall}\mathbf{GF} \tag{5.13}$$

[2]This is often referred to as a *repeated reachability* or *progress property*.

The expansion laws demonstrate that $\mathbf{FG}\phi$ is satisfied if and only if ϕ holds in the final state of the trace. The same is true for $\mathbf{GF}$. Both properties thus coincide in their semantics which makes them practically indistinguishable. As a consequence, particular care is advised when properties containing nested temporal operators are to be evaluated. In order to increase the expressiveness of LTL on finite traces, De Giacomo and Vardi proposed a new logic, *Linear-time Dynamic Logic* (LDL_f), which combines the convenience of LTL with the expressive power of regular expressions defined over finite traces (RE_f) [69]. They show that LDL_f is strictly more expressive than LTL_f and allows for the formulation of a range of property types which are not expressible in LTL_f (e.g. *alternating sequences*). We aim to further investigate the usefulness of LDL_f for the verification of correctness criteria of agent-based simulations as part of our future work (see Section 10.3.2).

Now that the general semantics of LTL on finite traces have been clarified, we can describe the syntax and semantics of simLTL, starting with the agent layer, in the next section.

5.3 The agent layer

The agent layer — the core of simLTL — represents an extension of conventional LTL which allows for the formulation of temporal properties about agent traces, i.e. finite sequences of agent states. Each well-formed simLTL agent formula $\phi : \Phi$ can be recursively defined as follows:

$$\begin{aligned} \phi ::= & \mathit{pred} \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\, \phi \mid \mathit{aval} \trianglelefteq \mathit{aval} \mid \\ & \mathbf{X}_\forall \phi \mid \mathbf{X}_\exists \phi \mid \phi\, \mathbf{U}\, \phi \mid \phi\, \mathbf{R}\, \phi \\ \mathit{aval} ::= & \mathit{att} \mid \mathit{aval} \oplus \mathit{aval} \mid \mathit{Value} \end{aligned}$$

where

- $\mathit{pred} : \mathit{Name} \times \mathbb{F}\ \mathit{Value}$ is a *predicate*,
- $\trianglelefteq \in \{<, >, \leq, \geq, =, \neq, \approx, \not\approx\}$ is a *comparison operator*
- $\mathit{att} : \mathit{Name}$ is an *attribute name*
- $\oplus \in \{+, -, *, /\}$ is an *arithmetic operator*

'$\mathbf{X}_\forall$', '$\mathbf{X}_\exists$', '$\mathbf{U}$' and '$\mathbf{R}$' are temporal operators called 'Weak Next', 'Strong Next', 'Until' and 'Release', respectively; the distinction between a strong and a weak version of 'next' is due to the evaluation of simLTL formulae on finite traces and has been discussed in more detail in Section 5.2.2; $\mathbf{X}_\forall\phi$ is true whenever ϕ holds in the next state; if there is no next state (because ϕ is being evaluated upon the final state of a trace), $\mathbf{X}_\forall\phi$ always evaluates to true. On non-final states, $\mathbf{X}_\exists\phi$ has the same semantics as $\mathbf{X}_\forall\phi$; when being evaluated on a final state, however, $\mathbf{X}_\exists\phi$ always evaluates to false. $\phi_1\ \mathbf{U}\ \phi_2$ is true if ϕ_1 holds in all subsequent states up to a future state in which ϕ_2 holds, and $\phi_1\ \mathbf{R}\ \phi_2$ is true if ϕ_2 always holds unless there is a future state until and including which ϕ_2 holds and starting from which ϕ_1 holds (ϕ_1 'releases' ϕ_2).

We further define the following equivalences:

$$\text{true} \equiv \phi \vee \neg\ \phi \tag{5.14}$$

$$\text{false} \equiv \phi \wedge \neg\ \phi \tag{5.15}$$

$$\phi_1 \Rightarrow \phi_2 \equiv \neg\ \phi_1 \vee \phi_2 \ \text{(implication)} \tag{5.16}$$

$$\phi_1 \Leftrightarrow \phi_2 \equiv (\phi_1 \Rightarrow \phi_2) \wedge (\phi_2 \Rightarrow \phi_1) \ \text{(equivalence)} \tag{5.17}$$

$$\phi_1 \mathbf{W} \phi_2 \equiv (\phi_1\ \mathbf{U}\ \phi_2) \vee \mathbf{G}\phi_1 \ \text{('weak until')} \tag{5.18}$$

$$\mathbf{F}\phi \equiv \text{true}\ \mathbf{U}\ \phi \ \text{('finally', 'eventually')} \tag{5.19}$$

$$\mathbf{G}\phi \equiv \neg\ \mathbf{F} \neg\ \phi \ \text{('globally', 'always')} \tag{5.20}$$

Although the 'release' operator can be defined in terms of 'until' (as shown in Equivalence 2.29), it has been deliberately made a first-class citizen of simLTL. This is necessary because of the translation of the formula into *positive normal form (PNF)* which requires a dual operator for each operator to be incorporated into the language ('release' is the dual operator of 'until'). The translation of simLTL formulae into PNF prior to verification is useful since it simplifies the evaluation. More details are given in Section 6.3.1.

The semantics of simLTL agent formulae are given on individual agent traces. For a single agent trace t_a of length k, the satisfaction of a simLTL formula ϕ (denoted by $t_a \models \phi$) can be defined as follows:

$$t_a \models (pred, X) \Leftrightarrow P_a(pred, t_a[0], X) = \text{true} \tag{5.21}$$

$$t_a \models \neg\ \phi \Leftrightarrow t_a \not\models \phi \tag{5.22}$$

$$t_a \models \phi_1 \wedge \phi_2 \Leftrightarrow t_a \models \phi_1 \text{ and } t_a \models \phi_2 \tag{5.23}$$

$$t_a \models \phi_1 \vee \phi_2 \Leftrightarrow t_a \models \phi_1 \text{ or } t_a \models \phi_2 \tag{5.24}$$

$$t_a \models \mathbf{X}_\forall\, \phi \Leftrightarrow \#t_a[1..] = 0 \vee t_a[1..] \models \phi \tag{5.25}$$

$$t_a \models \mathbf{X}_\exists\, \phi \Leftrightarrow \#t_a[1..] > 0 \wedge t_a[1..] \models \phi \tag{5.26}$$

$$t_a \models \phi_1\, \mathbf{U}\, \phi_2 \Leftrightarrow \exists 0 \leq i < k \bullet t_a[i..] \models \phi_2 \wedge (\forall 0 \leq j < i \bullet t_a[j..] \models \phi_1) \tag{5.27}$$

$$t_a \models \phi_1\, \mathbf{R}\, \phi_2 \Leftrightarrow \exists 0 \leq i < k \bullet t_a[i..] \models \phi_1 \wedge (\forall 0 \leq j \leq i \bullet t_a[j..] \models \phi_2)$$
$$\textbf{or } \forall 0 \leq i < k \bullet t_a[i..] \models \phi_2 \tag{5.28}$$

$$t_a \models aval_1 \trianglelefteq aval_2 \Leftrightarrow Eval(t_a[0], aval_1) \trianglelefteq Eval(t_a[0], aval_2) \tag{5.29}$$

Line 5.21 describes the satisfaction of a predicate which is determined by the predicate function P_a described in Section 4.2.2. All other definitions are straightforward, except the final one. Line 5.29 describes the semantics of a relation statement. Here, $Eval : AState \times Value \rightarrow Value$ denotes an evaluation function which accepts any local state $s_a : AState$ and any $v : Value$ as an input, evaluates v on s_a and returns a real number. If evaluated on an agent trace t_a, $val \trianglelefteq val$ thus holds if and only if it holds on the initial state, i.e. $Eval(t_a[0], val_1) \trianglelefteq Eval(t_a[0], val_2)$.

At this point, it is useful to briefly explore the relationship between simLTL and other extensions of LTL that allow for the specification of numeric constraints directly within the logical language [75]. Conventional LTL is propositional in nature, i.e. statements are made about atomic propositions whose truth or falsity can be determined at particular points in time. For example, proposition p may be true if and only if the following inequality holds $x < y$ (where x and y are both variables in the underlying system); p is thus true at time t if and only if the value of x at time t is less than the value of y at time t. In order to state that *"x is always less than y"* , we can write $\mathbf{G}p$. Propositions represent convenient abstractions that hide some of the complexities of the underlying system. In many cases, however, it may be more convenient to access the underlying values directly, e.g. such that x and y can be referred to from within the specification language. This is the purpose of temporal logics that provide support for constraints (Constraint-LTL or CLTL; see [75] for an overview of extensions of LTL with support for *Presburger constraints*). In general, these languages extend the syntax of LTL with an atomic formula of the form $R(\mathbf{X}^{l_1} x_{j_1}, ..., \mathbf{X}^{l_n} x_{j_n})$ where $x_i : D$ is a variable ranging over a numeric domain (typically integers or natural numbers), R is an n-ary relation, and $\mathbf{X}$ is the conventional temporal 'next' operator. CLTL allows for the formulation of constraints between values of variables *at different points in time*. For example, formula $\mathbf{G}(x < \mathbf{XX}y)$ states that *"the value of x is always less than the value of y two time steps later"*. simLTL provides access to numeric variables through *agent attributes* and *agent* and

group aggregation functions (see Section 4.2.1). It is, however, only possible to formulate relations between values *at the same point in time*; in that respect, simLTL is equivalent to CLTL with *zero-step constraints* (i.e. constraints over individual time points) and *binary relations* (i.e. $x < y$ but not $x < y < z$). Since, in the context of this work, simLTL formulae are only ever evaluated on given finite traces, satisfiability is decidable.

Another stream of work worth mentioning at this point is extensions of LTL and their model checking for the purpose of *constraint satisfaction* [90, 80] (see Section 2.4.2.2). Here, instead of determining whether a given model satisfies the constraints stated in a property, model checking is used is to find an appropriate *substitution* for unbound variables, i.e. to determine the *domain of validity* for making the given property true. In this context, model checking can thus be seen as being used in an 'inverted' sense, to *infer* a set of models that satisfy a given property. Constraints specifiable in simLTL may only contain reference to existing, bound variables; the substitution problem is therefore not an issue in this work.

Linear time properties are *path properties* by default, i.e. their semantics are defined over paths rather than over states. As a consequence, any agent trace t_a satisfies a simLTL agent formula ϕ if and only if ϕ is satisfied by the initial state of t_a, i.e. we have

$$t_a \models \phi \Leftrightarrow t_a[0] \models \phi \tag{5.30}$$

Agent formulae are useful for describing the behaviour of individual agents. However, since agents in a simulation do not live in isolation but rather as part of an entire population, we now need to extend our focus to the formulation of properties about groups of agents. This is the purpose of the *global layer* described in the next section.

5.4 The global layer

Agent formulae allow for the formulation of agent-centric properties. Since all properties are ultimately evaluated on full simulation traces (i.e. sequences of *group states*), however, we need to extend the logic appropriately such that it allows for the formulation of properties about *groups of agents*. The syntax of a full simLTL formula $\psi : \Psi$ is defined recursively as follows:

$$
\begin{aligned}
\psi ::= & \; pred \mid \phi^a \mid \psi \wedge \psi \mid \psi \vee \psi \mid \neg\, \psi \mid gval \trianglelefteq gval \\
& \mid \mathbf{X}_\forall \psi \mid \mathbf{X}_\exists \psi \mid \psi \,\mathbf{U}\, \psi \mid \psi \,\mathbf{R}\, \psi \\
& \mid \forall\, \psi \mid \exists^{\trianglelefteq\, k}\, \psi \mid \exists_r^{\trianglelefteq\, k}\, \psi \mid \langle\!\langle \phi \rangle\!\rangle\, \psi \\
gval ::= & \; gval \oplus gval \mid \#\langle\!\langle \phi \rangle\!\rangle \mid val \mid \mathbb{R}
\end{aligned}
$$

As denoted by the second value in the first line ('ϕ^a'), any agent formula $\phi : \Phi$ is also a full simLTL formula. Since simulation traces are composed of n agent traces, however, it needs to be defined *which* agent trace the formula is to be evaluated upon. This is denoted by the superscript 'a' which we refer to as the *agent in scope*; more details are given further below.

In addition to the Boolean and temporal operators described above, simLTL formulae also comprise a *selection operator* $\langle\!\langle \phi \rangle\!\rangle$ which is described in further detail in Section 5.4.2. $\forall$ and $\exists$ are quantifiers related to, yet slightly different from the ones in propositional logic. Their semantics are described in more detail in Section 5.4.1. The equivalences that were defined for agent formulae above also hold for full simLTL formulae.

simLTL formulae are evaluated on simulation traces. Let $t_s : Tr_s$ denote a simulation trace of length k. As indicated above, we need to augment the simulation trace with information about the *agent in scope* in order to allow for the formulation of properties about individual agents. Each simLTL formula is thus evaluated on a tuple (t_s, a) where $t_s : Tr_s$ is the simulation trace and $a : ids(t_s[0]) \cup \{rand\}$ is the agent in scope. As indicated by '$rand$', the agent in scope can be undefined and thus set to be randomly chosen. More details are given further below. The semantics of the non-temporal, non-quantified fragment of simLTL formulae are defined as follows:

$$(\varnothing, a) \models \psi \Leftrightarrow \text{false} \tag{5.31}$$

$$(t_s, a) \models (pred, X) \Leftrightarrow P_g(pred, t_s[0], X) = \text{true} \tag{5.32}$$

$$(t_s, a) \models \phi \Leftrightarrow a \in ids(t_s[0]) \wedge t_a \models \phi \tag{5.33}$$

$$(t_s, a) \models \neg\, \psi \Leftrightarrow (t_s, a) \not\models \psi \tag{5.34}$$

$$(t_s, a) \models \psi_1 \wedge \psi_2 \Leftrightarrow (t_s, a) \models \psi_1 \text{ and } (t_s, a) \models \psi_2 \tag{5.35}$$

$$(t_s, a) \models \psi_1 \vee \psi_2 \Leftrightarrow (t_s, a) \models \psi_1 \text{ or } (t_s, a) \models \psi_2 \tag{5.36}$$

As described by Equivalence 5.31, a simLTL formula evaluates to false whenever it is evaluated on an empty simulation trace. It is important to handle this case since an unsuccessful selection operation (see Section 5.4.2 below) always returns an empty simulation trace. Equivalence 5.33 describes the evaluation of an agent formula on a simulation trace. Each simulation trace is composed of n agent traces. When trying to evaluate an agent formula ϕ on a simulation trace, it is thus important to specify *which* agent trace the formula is to be evaluated upon. This information is stored in variable a which represents the agent in scope. If the variable is undefined (which is the case if the agent formula is not enclosed by a quantifier), then a value is chosen nondeterministically. Since true nondeterminism cannot be dealt with in a computational context, however, it will be resolved into a random choice. As formally motivated in Section 3.6, this has an important implication on the possibility to formulate properties about the *average behaviour* of agents.

The semantics for the remaining operators are given below:

$$(t_s, a) \models \mathbf{X}_\forall\, \phi \Leftrightarrow \#t_s[1..] = 0 \vee (t_s[1..], a) \models \phi \tag{5.37}$$

$$(t_s, a) \models \mathbf{X}_\exists\, \psi \Leftrightarrow \#t_s[1..] > 0 \wedge (t_s[1..], a) \models \psi \tag{5.38}$$

$$(t_s, a) \models \psi_1\ \mathbf{U}\ \psi_2 \Leftrightarrow \exists\, 0 \leq i < k \bullet (t_s[i..], a) \models \psi_2 \wedge (\forall\, 0 \leq j < i \bullet (t_s[j..], a) \models \psi_1) \tag{5.39}$$

$$(t_s, a) \models \phi_1\ \mathbf{R}\ \phi_2 \Leftrightarrow \exists\, 0 \leq i < k \bullet (t_s[i..], a) \models \psi_1 \wedge (\forall\, 0 \leq j \leq i \bullet (t_s[j..], a) \models \psi_2) \textbf{ or } \forall\, 0 \leq i < k \bullet (t_s[i..], a) \models \psi_2 \tag{5.40}$$

$$(t_s, a) \models \forall\, \psi \Leftrightarrow \forall\, a' : ids(t_s[0]) \bullet (t_s, a') \models \psi \tag{5.41}$$

$$(t_s, a) \models \exists^{\trianglelefteq n}\ \psi \Leftrightarrow \#\{a' : ids(t_s[0]) \mid (t_s, a') \models \psi\} \trianglelefteq n \tag{5.42}$$

$$(t_s, a) \models \exists_r^{\trianglelefteq k}\ \psi \Leftrightarrow \#\{a' : ids(t_s[0]) \mid (t_s, a') \models \psi\}/(\#t_s[0]) \trianglelefteq k \tag{5.43}$$

$$(t_s, a) \models \langle\!\langle \phi \rangle\!\rangle\ \psi \Leftrightarrow \exists\, ids : \mathbb{F}\ Value \mid (\forall\, a' : ids \bullet t_s[0][a'] \models \phi) \bullet (\mathit{filter}(t_s, ids), rand) \models \psi \tag{5.44}$$

$$(t_s, a) \models gval_1 \trianglelefteq gval_2 \Leftrightarrow Eval(t_s[0], gval_1) \trianglelefteq Eval(t_s[0], gval_2) \tag{5.45}$$

The temporal fragment of full simLTL formulae is analogous to the temporal fragment of agent formulae. Equivalence 5.41 gives another example of substitution by means of universal quantification. If a formula is enclosed by a universal quantifier, a is substituted once for each agent trace in the current

simulation trace. The semantics of the three quantifiers (Equivalences 5.41-5.43) are described in further detail in the next section; the semantics of the selection operator (Equivalence 5.44) are described in Section 5.4.2.

5.4.1 Quantification

As motivated in Section 4.3, when dealing with a (potentially large) group of agents, having the ability to quantify the number of agents for which a property is to hold is an important requirement. In the following sections, we describe how *universal* and *existential* quantification is represented in simLTL. It is important to note that quantification is mere syntactic sugar and does not increase the expressivity of the specification language. Since quantification is only defined for the countable and finite set of agents, the same statements could be expressed without quantification operators. As we show further below, however, quantified statements are linearly more succinct (in the universal case) and factorially more succinct (in the existential case) than unquantified ones. It is also important to note that both quantification and selection (see Section 5.4.2) have a significant impact on the time complexity of verification, raising it from linear to exponential in the size of the formula; this is described in more detail in Sections 6.2.2 and 6.3.6.

5.4.1.1 Universal quantification

In order to understand universal quantification, consider again Equivalence 5.41 given above:

$$(t_s, a) \models \forall\, \psi \Leftrightarrow \forall\, a' : ids(t_s[0]) \bullet (t_s, a') \models \psi$$

It states that a simulation trace satisfies $\forall\, \psi$ if and only if it satisfies ψ for any substitution of a with a', where $a' : ids(t_s[0])$ is a unique agent id. It is important to note that quantification is restricted to the set of agent ids; it is not possible to quantify over other values, e.g. agent attributes.

Example 2. Consider the following proposition: *'All male agents should finally be infected'*. In order to formulate this in simLTL, we assume that each agent has an attribute '$gender \in \{m, f\}$' as well as a predicate '$Infected$' which denotes whether the agent is infected or not. The property can then be formulated as follows:

$$\forall\,[gender = m \Rightarrow \mathbf{F}\ Infected] \tag{5.46}$$

This formula creates a separate version of the enclosed formula '$gender = m \Rightarrow \mathbf{F}\ Infected$' for each possible substitution of a with a value a' drawn from $ids(t_s[0])$, i.e. a unique agent id; the resulting formulae are then evaluated on the respective agent traces. Note that, in the example above, if the gender of the agents may change during the course of the simulation, then we need to enclose the statement into a 'globally' expression in order to make it a temporal invariant.

It is important to note that the universal quantifier is always succeeded by a full simLTL formula and not — as possibly expected — by an agent formula. This is necessary for the formulation of *multi-level properties* as described in Section 4.3, i.e. properties which contain both individual and group-level statements.

Example 3. Consider the following statement: 'If any agent is initially infected, then the entire population will eventually be infected'. The property can be formalised as follows:

$$\forall\,[Infected \Rightarrow \mathbf{F}(\forall\ Infected)] \tag{5.47}$$

The implication in this property is a full simLTL formula. The antecedent of the implication is an agent formula. The consequent is again a full simLTL formula with an agent formula wrapped inside. If the universal quantifier were succeeded by an agent formula instead, then the property above would not be expressible. Note that the consequent is an aggregate property. It makes a statement about the population *as a whole* since it requires reaching a global state in which *all agents* are infected. This differs from the following property:

$$\forall\,[Infected \Rightarrow \forall(\mathbf{F}\ Infected)] \tag{5.48}$$

In this case, the consequent is *not* aggregate since it requires *each agent separately* to reach a state in which it is infected. Again, the property would not be expressible if universal quantification were not succeeded by a group but by an agent formula instead. In this way, ambiguities as those shown by the example sentence '*eventually all male agents will know the new product*' in Section 4.3 can be resolved.

It is obvious that the same statements could have been formulated without the quantification operator, by means of conjunction. Consider, for example, the first statement above. Given a population of n agents, it could have been formulated without quantification as follows[3]:

$$
\begin{aligned}
&[(att(gender) = m) \Rightarrow \mathbf{F}\ Infected]^1 \wedge \\
&[(att(gender) = m) \Rightarrow \mathbf{F}\ Infected]^2 \wedge \\
&[(att(gender) = m) \Rightarrow \mathbf{F}\ Infected]^3 \wedge \\
&... \\
&[(att(gender) = m) \Rightarrow \mathbf{F}\ Infected]^n
\end{aligned}
$$

Intuitively, the negation of a 'forall' statement states that there exists at least one agent for which the given formula does *not* hold. The negation of a 'forall' formula can thus be expressed in terms of the existential quantifier whose logic is introduced in the following section:

$$\neg\ \forall \psi \equiv \exists^{\geq 1} \neg\ \psi \tag{5.49}$$

This relationship between universal and existential quantification is the reason why both operators are succeeded by the same type of formula (full simLTL formula). Otherwise the negation of 'forall' formulae would not be possible.

5.4.1.2 Existential quantification

When making statements about (potentially large) populations of individuals, it is important to be able to express quantities — both absolute and relative. For example, in an epidemiological model, we may want to express criteria like *'an agent can only become infected if at least 10 of its neighbours are infected'* or *'it should never be possible to reach a state in which more than 80% of all agents are infected'*. As shown below, quantitative expressions like *'at least 80%'* are non-trivial to formulate in conventional LTL. In fact, it requires the enumeration of all possible combinations of agents for which the statement is true, which results in combinatorial explosion — even for small populations.

[3]The superscript numbers denote the substitution of a (the agent in scope) for the respective subformula.

In order to facilitate the formulation of absolute and relative quantitative statements, we introduce a quantification operator $\exists^{\trianglelefteq k}$ where $\trianglelefteq \in \{<, >, \leq, \geq, =\}$ and $k \in \mathbb{N}$. Let us first look at absolute quantification. Given a population of n agents, we need a way of formulating a statement like *'for exactly k out of n agents, ψ holds'* without having to enumerate and evaluate all $\binom{n}{k}$ possible combinations of individual agents exhaustively. To this end, we introduce the first shortcut — the *absolute existential quantifier*:

$$\exists^{=k}(\psi) \equiv \bigvee_{c \in \binom{N}{k}} \bigwedge_{a' \in c} (t_s, a') \models \psi \tag{5.50}$$

Here and in all subsequent equations, we denote with N the set of all n agents. Formula 5.50 illustrates that the absolute quantifier is significantly more succinct than the disjunction of the conjunction otherwise necessary. The usefulness of a shortcut is even more apparent in case of range-based quantification, i.e. if we are not interested in a single number but rather in a range of agents for which the whole group satisfies a formula ψ, as exemplified using '$\leq$' below:

$$\exists^{\leq k}(\psi) \equiv \bigvee_{m=0}^{k} \bigvee_{c \in \binom{N}{m}} \bigwedge_{a' \in c} (t_s, a') \models \psi \tag{5.51}$$

Finally, conventional logic gets most unwieldy when range-based **and** relative quantitative statements need to be made, i.e. range-based statements about *percentages* of agents. To this end, we introduce a *relative existential quantifier* (denoted by the subscript letter 'r'), exemplified using '$\geq$' below:

$$\exists_r^{\geq x}(\psi) \equiv \bigvee_{m=\lceil x \cdot n \rceil}^{n} \bigvee_{c \in \binom{N}{m}} \bigwedge_{a' \in c} (t_s, a') \models \psi \tag{5.52}$$

where $0 < x \leq 1$ and n denotes the total number of agents in the current simulation trace. It is clearly apparent that, in this case, the conventional logical notation is entirely impractical.

The negation of existential quantifiers is given below:

$$\neg\ \exists^{\trianglelefteq k}\ \psi \equiv \exists^{\neg\ (\trianglelefteq k)}\ \psi \tag{5.53}$$

$$\neg\ \exists_r^{\trianglelefteq k}\ \psi \equiv \exists_r^{\neg\ (\trianglelefteq k)}\ \psi \tag{5.54}$$

The negation of the comparison statement is defined as follows:

$$\neg\left(v_1 < v_2\right) \equiv v_1 \geq v_2 \tag{5.55}$$

$$\neg\left(v_1 \leq v_2\right) \equiv v_1 > v_2 \tag{5.56}$$

$$\neg\left(v_1 = v_2\right) \equiv v_1 \neq v_2 \tag{5.57}$$

$$\neg\left(v_1 \geq v_2\right) \equiv v_1 < v_2 \tag{5.58}$$

$$\neg\left(v_1 > v_2\right) \equiv v_1 \leq v_2 \tag{5.59}$$

5.4.2 Selection

So far, on a global level, simLTL allows for the formulation of three types of properties: aggregate group properties and universally and existentially quantified agent properties. An example of the first type is the following property: *'The average income of all agents will never fall below x'*. An example of the latter type is the following property: *'the system will never reach a global state in which all agents have an income lower than X'*. As described in Section 4.3, however, it is often necessary to constrain the focus of interest to a single agent or a particular group of agents. To this end, we introduce a *selection operator* which allows us to constrain the focus to a subset of all agents based on a certain criterion. A selection operator has the form $\langle\!\langle \phi \rangle\!\rangle$ where $\phi \in \Phi_a$ is an *atemporal* simLTL formula (an agent formula which does not contain any temporal operators) that acts as a *selection criterion*. Restricting the selection to atemporal formulae is necessary in order to keep the evaluation of properties (described in Chapter 6) efficient.

Formally, selection can be defined as a function that maps from the *superordinate set of agent traces*, i.e. those agent traces which are part of the simulation trace that the current formula is being evaluated upon, to one of its subsets. More precisely, if a selection operator is applied to a simulation trace t_s then the result is another simulation trace comprising exactly those agent traces t_a which satisfy the given criterion. This is formally expressed below:

$$\langle\!\langle \phi \rangle\!\rangle = t_s : Tr_s \text{ such that } \forall\, t_a : traces_a(t_s) \bullet t_a \models \phi$$

Selection is best viewed as a constraint. By using a selection operation, the desired validity of a simLTL formula is constrained to a subset of the overall agent population. Statements involving selections thus

allow for the expression of both individual and (universal and existential) group properties. Since the agent which is currently in scope may not be a member of the group resulting from a selection, it is important that it is reset to 'rand' as indicated in Equivalence 5.44 above.

Example 4. Consider the following proposition: *'All male agents should finally be infected'*. In order to formulate this in simLTL, we assume that each agent has an attribute $gender \in \{m, f\}$ which can be assigned values from the set $\{m, f\}$. The property above can then be formulated as follows:

$$\langle\!\langle gender = m \rangle\!\rangle \; \forall(\mathbf{F}\; Infected) \tag{5.60}$$

This property expresses that '*within the group of male agents, all agents will finally be infected*'. This property can also be expressed without selection as follows:

$$\forall(gender = m \Rightarrow \mathbf{F}\; Infected) \tag{5.61}$$

For individual agent properties, selection is thus equivalent to logical implication. For full simLTL formulae, however, this equivalence does not hold. Consider, for example, the following statement: '*the group of all male agents will finally have an average income of x*'. Since the second part of the property ('*will finally have an average income of x*') is a full simLTL formula, it is impossible to express the same statement using universal quantification and logical implication as in Property 5.61. Instead, it needs to be split into a selection part (which defines the group of agents that the second part of the property is evaluated upon) and the actual evaluation of the second part of the property as shown below. Note that $avgIncome$ represents a nullary numeric function which calculates and returns the average income based on the individual agent incomes.

$$\langle\!\langle gender = m \rangle\!\rangle \; \mathbf{F}(avgIncome = x) \tag{5.62}$$

Whereas in the individual case, selection can thus be seen as mere 'syntactic sugar', it is strictly necessary in the presence of a full simLTL formula. Note that all three properties above are semantically different from the following one:

$$\forall\,(gender = m \;\wedge\; \mathbf{F}\; Infected) \tag{5.63}$$

Property 5.63 asserts that all agents are male and will finally be infected — a statement which is definitely false as soon as there is at least a single female agent in the population. In contrast, Properties 5.60 – 5.63 constrain the focus of interest before the actual evaluation takes place. That is, it checks the validity of predicate $Infected$ only on male agents.

As mentioned above, a simLTL formula is always evaluated on those agent traces which are currently in scope. On the topmost level, this is the set of all agent traces in the population. In the example above, the selection operation will thus restrict the scope by selecting all agents which satisfy the simLTL formula $gender = m$. Since this is a state formula and it is evaluated on a path, all agents whose $gender$ attribute is m in the initial state are selected. This is not a problem unless agents are capable of changing their gender during the simulation. In this case, evaluation on the initial path only would not suffice and the property would, for example, have to be reformulated in a temporal way. Property $\mathbf{F}\ infected$ is then evaluated on the resulting set of traces. The formula is precisely true if it holds for all agent traces that it is being evaluated upon, i.e. for all male agents. It is, of course, perfectly possible that a selection operation is unsuccessful, i.e. that the result of a selection operation is empty. In this case (as described in the semantics of full simLTL formulae above), the subsequent formula always evaluates to false.

If a simLTL formula comprises constituents which are themselves simLTL formulae, then each of them can be individually preceded by a selection criterion, as shown in the following example.

Example 5. Consider the following expression: 'If all male agents below the age of 25 are infected, then all male agents will eventually be infected'. This can be formulated as follows:

$$\mathbf{G}\,[\langle\!\langle gender = m \wedge age < 25 \rangle\!\rangle\ \forall\, Infected \Rightarrow \mathbf{F}(\langle\!\langle gender = m \rangle\!\rangle\ \forall\, Infected)]$$

This formula reads as follows: '*It is always the case that if, within the group of male agents below the age of 25, everyone is infected, then eventually, within the group of male agents, everyone will be infected*'. At this point, it is important to mention the high precedence of the selection operator; in the example above, both selection operations are applied to their immediately succeeding simLTL formulae only. If a selection operation is to be applied to a more complex, non-atomic formula ϕ, then ϕ needs to be enclosed in parentheses as illustrated in the following example. As mentioned above, selection always happens based on the current superordinate set of agent traces. By nesting selection criteria, complex expressions in which a set of agent traces is successively filtered can be formulated. This is illustrated by the following example.

Example 6. Consider the following expression: 'It is true for all male agents which are susceptible that, if they are not informed appropriately (denoted by the predicate $Informed$), then those which are older than 20 will finally be infected.' This can be formulated as follows:

$$\langle\!\langle gender = m \wedge Susceptible \rangle\!\rangle \, [\neg \, Informed \Rightarrow \langle\!\langle age > 20 \rangle\!\rangle \; \forall(\mathbf{F} \, Infected)]$$

Note that, in this example, the second selection operation $\langle\!\langle age > 20 \rangle\!\rangle$ is applied to the set of agent traces which results from the first selection $\langle\!\langle gender = m \wedge Susceptible \rangle\!\rangle$.

It remains to describe the negation of selection statements. Informally, the negation of a selection statement states that the group denoted by the selection criterion does not satisfy the given formula. This is described by the following equivalence:

$$\neg \, (\langle\!\langle \phi \rangle\!\rangle \; \psi) \equiv \langle\!\langle \phi \rangle\!\rangle (\neg \; \psi) \tag{5.64}$$

This completes the description of the semantics of simLTL. The next section briefly illustrates how complex custom logic (as, for example, needed during model validation) can be embedded into a simLTL-based verification framework.

5.5 Integrating external logic

Any serious attempt at external validation of an agent-based simulation will invariably involve a certain amount of statistical analysis. Despite this work's focus on correctness analysis, it is not our intention to develop a general purpose language for simulation output analysis. Instead, with simLTL, we aim to construct a *declarative* language which describes correctness criteria in a simple and intuitive way. Nevertheless, certain calculations are necessary in order to answer particular properties. Consider, for example, a property which states that the average income of all agents should never fall below a certain value. In order to answer this property, the average needs to be calculated. Where should that be done? We certainly do not want to integrate the description of the average calculation into the property.

In order to solve this problem, we propose a *separation of concerns* with respect to the declarative and the procedural components of the property specification process. Declarative components are those which describe the basic, desired temporal behaviour of the system under consideration. Procedural

components are responsible for providing additional logic required for certain calculations at runtime. The syntax simLTL defines the declarative part of the language — the 'logical view' on the property. Procedural components are formulated in a general purpose programming language and contain external logic, e.g. the calculation of a certain statistical function such as the *squared error*, as motivated in Section 4.4.

In order to realise the integration of external logic, predicates are interpreted as calls to Boolean functions and numeric variables are interpreted as calls to numeric functions. The logic of both functions is customisable and can be formulated in a high-level programming language in the underlying simulation framework. Further details regarding the implementation are given in Chapter 8. The interpretation of atomic propositions and numeric values as function calls realises the aforementioned separation of concerns. simLTL is responsible for the declarative part of the property specification. For those cases in which a prescriptive, more *operational* description of certain aspects (e.g. statistical functions) is necessary, this logic can be encapsulated into a custom function. This preserves modularity and prevents the logical property from being cluttered with operational low-level details.

This is best illustrated with an example. Consider a complex simulation in which agents may freely roam through a two-dimensional space, consistently forming and breaking relationships with other agents based on their spatial proximity. Further assume that, based on its internal logic, every agent is only allowed to communicate with a maximum number of five neighbours at a time. Within the simulation, each agent keeps a record of who it is talking to in each time step. For efficiency reason, this information is stored in a hash map which is indexed by the time step. The monitor is not aware of the internals of the simulation; all it requires to perform the evaluation of a formula is a simulation trace which is itself ultimately composed of individual agent states — simple key-value pairs.

In order to formulate properties about the number of individual communication events, however, this information needs to be accessible from within simLTL. To that end, we can define a nullary numeric function $numNB : \mathbb{N}$ which encapsulates the parsing and counting logic and simply returns the number of neighbours. $numNB$ can then be used just like any other numeric constant within simLTL and the final property can be formulated as follows:

$$\mathbf{G}\,[\forall(numNB \leq 5)] \tag{5.65}$$

The most straightforward application of numeric functions and one that has been used frequently in the examples above is access to an agent's attributes. Another example of the encapsulation of (possibly complex) symbol manipulation is a unary predicate $Know : Ag \rightarrow \{\text{true}, \text{false}\}$ which can be used to check whether two agents know each other. Instead of integrating the operational information about discovering an agent's neighbourhood into the specification language itself, it is wrapped into a simple Boolean predicate function which returns true if the agent passed as an argument and the agent currently in scope are connected, otherwise false. In that way, properties that involve complex calculations (e.g. statistical functions) can be formulated in a succinct and descriptive way.

Another example is abstraction. It is often not convenient to represent agent attribute values exactly as they are in the property specification language. Imagine a scenario in which the age of an agent is represented as a natural number ranging from 0 to 100. For the purpose of validation, however, we may only be interested in certain age bands, for example band 1 (0–17 years), band 2 (18–65) and band 3 (66–100). This mapping is also best outsourced into a numeric function.

Other typical examples of incorporating prescriptive external logic are the use of numeric aggregation functions (sum, average, minimum, etc.) and the integration of external reference data with which the simulation output is to be compared in order to show its representational accuracy, as described in Section 4.4.

In order to exemplify the application of simLTL, the following section provides several typical correctness properties which are based on the simple diffusion model introduced in Section 5.1.1 above.

5.6 Example properties

The first set of example properties about the transmission model described in Section 5.1.1 concerns the individual agents' state transitions. As described above, agents are only allowed to transitions between the three states S (susceptible), I (infected) and R (recovered) as follows: $S \rightarrow I \rightarrow R$. Certain variants also allow the agents to transition back into S from R. These expectations can be formulated as invariants in simLTL as follows:

$$\forall\, \mathbf{G}\, \big(\mathit{Susceptible} \Rightarrow \neg\, \mathbf{X}_{\exists}\, \mathit{Recovered}\big) \tag{5.66}$$

$$\forall\, \mathbf{G}\, \big(\mathit{Recovered} \Rightarrow \neg\, \mathbf{X}_{\exists}\, \mathit{Susceptible}\big) \tag{5.67}$$

Note that, in this example, the strong version of the 'next' operator has to be chosen which is due to the enclosement of the entire formula into the 'globally' operator. In order for the semantics to be correct on the final state of a trace, the conclusion of the implication is required to evaluate to true. This is guaranteed if the strong version of 'next' is used. Note that this is only true because the conclusion of the implication contains a negation. Consider, for example, the following property which states that an agent, once recovered, will always remain so.

$$\forall\, \mathbf{G}\, \big(\mathit{Recovered} \Rightarrow \mathbf{X}_{\forall}\ \mathit{Recovered}\big) \tag{5.68}$$

Here, the weak version of the 'next' operator is required in order for the property to be evaluated correctly on the final state of a trace. The choice of the correct version of 'next' is clearly inconvenient. An alternative, less error-prone way of answering the properties above is to remove 'globally' and evaluate the property on trace fragments of length 2 (as formally motivated in Section 3.6). This prevents evaluation from ever taking place on a final state, as a consequence of which the problem mentioned above cannot arise and both the weak and the next version can be used interchangeably (denoted by a simple unquantified '$\mathbf{X}$'):

$$\forall\, (\mathit{Susceptible} \Rightarrow \neg\, \mathbf{X}\ \mathit{Recovered}) \tag{5.69}$$

$$\forall\, (\mathit{Recovered} \Rightarrow \neg\, \mathbf{X}\ \mathit{Susceptible}) \tag{5.70}$$

$$\forall\, (\mathit{Recovered} \Rightarrow \mathbf{X}\ \mathit{Recovered}) \tag{5.71}$$

Since the focus of this chapter is on the formulation of properties about single traces, the usage of trace fragments of different length is only mentioned informally here. Further information regarding the actual mechanism based on which fragments are extracted from full traces is given in Chapter 6.

The next property concerns the correct triggering of state transitions. According to our rules, an agent should only become infected if at least 80% of its neighbours are infected. Using the transition operator, selection, existential quantification, and the predicate $\mathit{Know} : \mathit{Ag} \rightarrow \{\text{true}, \text{false}\}$ introduced above, and evaluation on fragments of length 2, this can be formalised as follows:

$$\forall \Big[(\mathit{Susceptible} \wedge \mathbf{X}\ \mathit{Infected}) \Leftrightarrow \langle\!\langle \mathit{Know}(a) \rangle\!\rangle (\exists_r^{\geq 0.8}\ \mathit{Infected})\Big] \tag{5.72}$$

The formula states that '*it is always true for all agents that a transition from S to I occurs if and only if within the group of agents that know agent a, 80% are infected*'. Another criterion that we may want to ascertain is that, at any time, at most 5 of an agent's neighbours are infected. Using 'globally' and evaluation on full simulation traces, this can, for example, be formulated as follows:

$$\mathbf{G}\left(\forall\left[\exists^{\leq 5}(Know(a) \wedge Infected)\right]\right) \tag{5.73}$$

The formula states that '*it is always true for each agent a that there exist at most 5 other agents which know agent a and are infected*'. Alternatively, 'globally' can also be replaced with evaluation on fragments of length 1.

When dealing with populations of interacting agents, it can be important to check the correctness of social mechanisms, for example of information exchange between agents. A simple example could be that, if an agent is infected, then in the next step, at least 20% of its neighbours will also be infected. Evaluating on fragments of length 2, this can be formulated as follows:

$$\forall\left[Infected \Rightarrow \langle\!\langle Know(a) \rangle\!\rangle\ \exists_r^{\geq 0.2}(\mathbf{X}\ Infected)\right] \tag{5.74}$$

All previous criteria have been individual in nature and therefore universally quantified over the population. We now shift the focus to group-level properties. A typical criterion could be to prohibit reachability of a state in which all agents are infected:

$$\neg\ \mathbf{F}(\forall(Infected)) \tag{5.75}$$

Note that this formula is aggregate in nature. It thus differs from property $\forall\,[\neg\ \mathbf{F}(infected)]$ which ascertains *for each agent individually* that it will never reach a state in which it is infected.

The next property states that it is not possible to reach a state in which more than 5,000 agents are infected:

$$\neg\ \mathbf{F}(\exists^{>5000}(Infected)) \tag{5.76}$$

And finally, the following property states that the average age of all infected agents is always above 50:

$$\mathbf{G}(\langle\!\langle Infected \rangle\!\rangle (avgAge > 50)) \tag{5.77}$$

We hope to show with these examples that simLTL statements are reasonably close to their informal counterparts in natural language and thus more easily accessible for people without a background in mathematical logic than conventional propositional and temporal logic.

5.7 Summary

This chapter described simLTL, a *LTL-based property specification language* tailored to the formulation of *correctness properties* about agent-based simulation traces. In addition to conventional Boolean and temporal operators, simLTL offers the possibility to constrain subformulae to particular *groups of agents*, to make *quantitative statements about groups* of agents and to express *simple arithmetic relations*. *Custom external logic* is integrated by means of interpreting atomic predicates as well as numeric constants as calls to *Boolean* or *numeric functions*, respectively. This facilitates access to *agent attributes*, the formulation of *aggregation operations* and the inclusion of *external reference data*, e.g. historical time series data. In order to facilitate the realisation of *multi-level properties*, the syntax of simLTL is subdivided into an agent and a global level. Since simLTL formulae are evaluated on finite traces, a special semantics is required. Building upon ideas from related work, this was achieved by distinguishing between a weak and a strong version of the 'next' operator.

The next chapter presents an algorithm for the verification of simLTL properties on individual simulation traces as well as an estimation procedure for determining the approximate probability of a property being true in a given set of simulation trace fragments. Due to its online nature, the algorithm guarantees to stop and return a result as soon as the formula has either been satisfied or refuted. By interweaving evaluation and simulation, the amount of computation needed can be reduced to a minimum.

Chapter 6

Verifying simLTL properties

6.1 Introduction

The previous chapter described simLTL, a LTL-based specification language tailored to the formulation of correctness criteria about agent-based simulation traces. In general, the verification of LTL formulae is often reduced to a *language containment problem* and solved using automata theoretic techniques (see Section 2.4.3). This requires both the system description and the property of interest to be encoded as a finite state automaton which, given the complexity of agent-based simulations, is not feasible in this case. Motivated by three typical characteristics of agent-based simulations — randomness, temporal boundedness, and non-safety criticality — this work takes a different approach which we refer to as *statistical runtime verification*. As opposed to a formal model, the approach operates directly upon the running system which regularly emits information about its current state. This information is received by a *monitor* which observes the system and makes a real-time decision about its correctness. Due to its focus on sets of individual traces, the approach is inherently approximate. However, by using a statistical estimation procedure, accuracy and confidence of the verification results can be adjusted as necessary and clearly quantified. This approach has a number of benefits:

- Since the agent-based simulation does not need to be represented formally, the approach is applicable to arbitrary types of simulations, and it scales well with the size of the underlying population.

- Due to its online nature, the approach is efficient such that it stops as soon as a property is clearly satisfied or violated.
- With respect to multiple simulation traces, the algorithm is 'anytime', i.e. it will always produce a result, no matter how many traces a formula is being evaluated upon. More evaluations, however, will improve the precision of the result.

A central task in the context of runtime verification is the implementation of an efficient monitor, a program which runs in parallel with the system to be tested, observes its progress and checks for compliance with a given property. This chapter describes an evaluation algorithm which implements such a monitor and checks simulation traces for compliance with a given simLTL property. The 'model' that the algorithm operates upon is a simulation trace as defined in Section 3.5. It is important to note, however, that the notion of a 'trace' used in this chapter does not necessarily correspond with a full simulation trace, but rather with a trace fragment, as formally introduced in Section 3.6. The algorithm consists of two steps: (i) the evaluation of a simLTL formula on a given trace fragment, and, (ii) the approximation of a formula's probability against the background of the entire state space.

The first sections of this chapter are devoted to the first step. The first algorithm described in Section 6.2 provides a simple, general but also non-optimal solution to the problem of evaluating a simLTL formula on a given trace. The algorithm is exhaustive in nature; it constructs a parse tree of the formula to be verified and evaluates each of its subformulae recursively. It is thus well suited for '*a-posteriori* verification', i.e. for the verification of already existing traces which have been obtained elsewhere (e.g. from a simulation which is not under the control of the person performing the verification) [121]. The main focus of this chapter is on Section 6.3 which describes an optimised online version of the monitoring algorithm. It is based on the ideas of *temporal testers* [136] described in Section 2.4.4. It allows for the evaluation of a simLTL formula upon finite traces 'on-the-fly', i.e. in parallel to the simulation execution. The algorithm is *optimal* such that it guarantees to return a result as soon as possible. This is achieved through *laziness*, according to which the monitor triggers the simulation to advance if and only if the property has not already been either satisfied or violated.

Finally, Section 6.4 addresses the second step of the verification problem. It summarises the idea of approximating the truth of a simLTL formula through random sampling from the underlying simulation by means of repeated simulation. As described in Section 3.6, different properties require trace fragments of different length. The formula is then evaluated on each trace fragment using the techniques described in the first two sections. The ratio of successful and non-successful evaluations represents

an approximation of the formula's probability. For the calculation of the number of trace fragments necessary to achieve a desired level of accuracy, we use a simple algorithmic procedure which makes direct use of the Binomial distribution.

6.2 An offline evaluation algorithm

Given a simulation trace $t_s : Tr_s$, the evaluation problem for a simLTL formula ψ is to determine whether t_s satisfies ψ, denoted $t_s \models \psi$. One way to approach this is to follow the idea of *SAT-based bounded model checking* and construct a propositional formula which is satisfied if and only if t_s satisfies ψ [31]. Another way would be to follow the automaton-based approach used for the verification of LTL properties [232]. Since the primary purpose of this section is to serve as a preliminary step towards the definition of the more efficient online evaluation algorithm described in Section 6.3, we follow a different approach and describe a simple evaluation procedure which operates directly upon the simulation trace. It is analogous to the one for conventional branching time logic and comprises two steps [62]:

1. Let $Sub(\psi)$ denote the set of all subformulae of ψ. For each subformula $\Psi \in Sub(\psi)$, determine the *satisfaction set* $\texttt{Sat}$, i.e. the set of all those states $s \in t_s$ that satisfy Ψ.
2. t_s satisfies ψ if and only if $t_s[0] \in \texttt{Sat}$.

A brief description of a simple offline algorithm is given below.

6.2.1 Algorithm description

The first step is accomplished by a recursive labelling function which, starting with ψ determines the satisfaction set for each subformula of ψ. It performs the check in an *offline* or *a posteriori* manner, in which case the individual simulation traces are expected to exist prior to verification. The offline nature of the algorithm makes it unsuitable as a monitor and thus for true runtime verification. This limitation is removed in the *online* version of the algorithm which is described in Section 6.3.

The algorithm comprises two separate labelling functions $\texttt{SAT}_a$ and $\texttt{SAT}_g$ for agent and full simLTL formulae, respectively. An outline of the algorithm for $\texttt{SAT}_a$ is shown in Algorithm 3. Given agent trace $t_a : Tr_a$ and simLTL agent formula ϕ, it performs a depth-first search through the parse tree of

Algorithm 3 Outline of labelling function $\texttt{SAT}_a$ for agent formulae

Require: Agent trace t_a of length k, simLTL agent formula ϕ

1: **if** ϕ is $(pred, X)$ **then return** $\{t_a[i] \mid P_a(pred, t_a[i], X) = \text{true}\}$
2: **if** ϕ is $\neg\, \phi_1$ **then return** $\{t_a[i] \mid t_a[i] \notin \texttt{SAT}_a(t_a, \phi_1)\}$
3: **if** ϕ is $(\phi_1 \wedge \phi_2)$ **then return** $\{t_a[i] \mid t_a[i] \in \texttt{SAT}_a(t_a, \phi_1) \wedge t_a[i] \in \texttt{SAT}_a(t_a, \phi_2)$
4: **if** ϕ is $(\phi_1 \vee \phi_2)$ **then return** $\{t_a[i] \mid t_a[i] \in \texttt{SAT}_a(t_a, \phi_1) \vee t_a[i] \in \texttt{SAT}_a(t_a, \phi_2)$
5: **if** ϕ is $(av_1 \trianglelefteq av_2)$ **then return** $\{t_a[i] \mid t_a[i] \models \texttt{Eval}_a(t_a[i], av_1) \trianglelefteq \texttt{Eval}_a(t_a[i], av_2)\}$
6: **if** ϕ is $\mathbf{X}_\forall \phi_1$ **then return** $\{t_a[k]\} \cup \{t_a[i] \mid i < k \wedge t_a[i+1] \in \texttt{SAT}_a(t_a, \phi_1)\}$
7: **if** ϕ is $\mathbf{X}_\exists \phi_1$ **then return** $\{t_a[i] \mid i < k \wedge t_a[i+1] \in \texttt{SAT}_a(t_a, \phi_1)\}$
8: **if** ϕ is $(\phi_1\ \mathbf{U}\ \phi_2)$ **then return**
$\{t_a[i] \mid t_a[i] \in \texttt{SAT}_a(t_a, \phi_2) \vee (t_a[i] \in \texttt{SAT}_a(t_a, \phi_1) \wedge t_a[i] \in \texttt{SAT}_a(t_a, \mathbf{X}_\exists(\phi_1\ \mathbf{U}\ \phi_2)))\}$
9: **if** ϕ is $(\phi_1\ \mathbf{R}\ \phi_2)$ **then return**
$\{t_a[i] \mid t_a[i] \in \texttt{SAT}_a(t_a, \phi_2) \wedge (t_a[i] \in \texttt{SAT}_a(t_a, \phi_1) \vee t_a[i] \in \texttt{SAT}_a(t_a, \mathbf{X}_\forall(\phi_1\ \mathbf{R}\ \phi_2)))\}$

Algorithm 4 Outline of labelling function $\texttt{SAT}_g$ for full simLTL formulae

Require: Simulation trace t_s of length k, simLTL formula ψ

1: **if** ψ is $(pred, X)$ **then return** $\{t_s[i] \mid P_g(pred, t_s[i], X) = \text{true}\}$
2: **if** ψ is ϕ **then return** $\{t_s[i] \mid (\exists\, t_a : traces_a(t_s) \mid t_a[i] \in \texttt{SAT}_a(t_a, \phi))\}$
3: **if** ψ is $\neg\, \psi_1$ **then return** $\{t_s[i] \mid t_s[i] \notin \texttt{SAT}_g(t_s, \psi_1)\}$
4: **if** ψ is $(\psi_1 \wedge \psi_2)$ **then return** $\{t_s[i] \mid t_s[i] \in \texttt{SAT}_g(t_s, \psi_1) \wedge t_s[i] \in \texttt{SAT}_g(t_s, \psi_2)$
5: **if** ψ is $(\psi_1 \vee \psi_2)$ **then return** $\{t_s[i] \mid t_s[i] \in \texttt{SAT}_g(t_s, \psi_1) \vee t_s[i] \in \texttt{SAT}_g(t_s, \psi_2)$
6: **if** ψ is $(gv_1 \trianglelefteq gv_2)$ **then return** $\{t_s[i] \mid t_s[i] \models \texttt{Eval}_g(t_s[i], gv_1) \trianglelefteq \texttt{Eval}_g(t_s[i], gv_2)\}$
7: **if** ψ is $\mathbf{X}_\forall \psi_1$ **then return** $\{t_s[k]\} \cup \{t_s[i] \mid i < k \wedge t_s[i+1] \in \texttt{SAT}_g(t_s, \psi_1)\}$
8: **if** ψ is $\mathbf{X}_\exists \psi_1$ **then return** $\{t_s[i] \mid i < k \wedge t_s[i+1] \in \texttt{SAT}_g(t_s, \psi_1)\}$
9: **if** ψ is $(\psi_1\ \mathbf{U}\ \psi_2)$ **then return**
$\{t_s[i] \mid t_s[i] \in \texttt{SAT}_g(t_s, \psi_2) \vee (t_s[i] \in \texttt{SAT}_g(t_s, \psi_1) \wedge t_s[i] \in \texttt{SAT}_g(t_s, \mathbf{X}_\exists(\psi_1\ \mathbf{U}\ \psi_2)))\}$
10: **if** ϕ is $(\psi_1\ \mathbf{R}\ \psi_2)$ **then return**
$\{t_s[i] \mid t_s[i] \in \texttt{SAT}_g(t_a, \psi_2) \wedge (t_s[i] \in \texttt{SAT}_g(t_s, \psi_1) \vee t_s[i] \in \texttt{SAT}_g(t_s, \mathbf{X}_\forall(\psi_1\ \mathbf{R}\ \psi_2)))\}$
11: **if** ψ is $\langle\!\langle \phi \rangle\!\rangle \psi$ **then return**
$\{trs : \mathbb{P}\, traces_a(t_s) \mid (\forall\, t_a : trs \bullet t_a[i] \in \texttt{SAT}_a(t_a, \phi)) \bullet t_s[i] \mid t_s[i] \in \texttt{SAT}_g(t_s, \psi)\}$
12: **if** ψ is $\forall\, \psi$ **then return** $\{t_s[i] \mid (\forall\, s' \in t_s[i] \bullet s \in \texttt{SAT}_g(t_s, \psi))\}$
13: **if** ψ is $\exists^{\trianglelefteq k}\, \psi$ **then return** $\{t_s[i] \mid \#\{s \in t_s \mid s \in \texttt{SAT}_g(t_s, \psi)\} \trianglelefteq k\}$
14: **if** ψ is $\exists_r^{\trianglelefteq k}\, \psi$ **then return** $\{t_s[i] \mid \#\{s \in t_s \mid s \in \texttt{SAT}_g(t_s, \psi)\}/\#t_s[i] \trianglelefteq k\}$

ϕ and, for each subformula ϕ_s of ϕ, returns all states that satisfy ϕ_s. Agent trace t_a can thus be said to satisfy property ϕ if and only if its initial state satisfies ϕ, i.e. iff $t_a[0] \in \texttt{SAT}_a(t_a, \phi)$. Arithmetic expressions are evaluated using a special function $\texttt{Eval}_a : \mathit{AState} \times \mathit{Value} \to \mathit{Value}$ which accepts an agent state s and a value $val : \mathit{Value}$ as input, evaluates val on s and returns another member of Value. An outline of the algorithm for full simLTL formulae is given in Algorithm 4.

6.2.2 Time complexity

In order to determine the complexity of the offline evaluation algorithm described above, we rely on the idea of *recurrence relations*. For each (recursive) operation defined above, we define a recursive function which describes the computational steps necessary to carry out the respective operation. In order to solve the recurrence relations, we then try to find a 'closed form', i.e. a representation in which the recursion is eliminated, for example by unfolding.

We start the analysis with the evaluation of agent formulae. The recurrence relation for evaluating agent formulae ϕ on traces of length t, denoted $A(\phi, t)$, is given below:

$$A(pred, t) = t \cdot c \tag{6.1}$$

$$A(\neg\, \phi, t) = t \cdot c + A(\phi, t) \tag{6.2}$$

$$A(\phi_1 \wedge \phi_2, t) = t \cdot c + A(\phi_1, t) + A(\phi_2, t) \tag{6.3}$$

$$A(\phi_1 \vee \phi_2, t) = t \cdot c + A(\phi_1, t) + A(\phi_2, t) \tag{6.4}$$

$$A(aval_1 \trianglelefteq aval_2, t) = t \cdot c \tag{6.5}$$

$$A(\mathbf{X}\, \phi, t) = t \cdot c + A(\phi, t) \tag{6.6}$$

$$A(\phi_1\, \mathbf{U}\, \phi_2, t) = t \cdot c + A(\phi_1, t) + A(\phi_2, t) \tag{6.7}$$

$$A(\phi_1\, \mathbf{R}\, \phi_2, t) = t \cdot c + A(\phi_1, t) + A(\phi_2, t) \tag{6.8}$$

We assume that the evaluation of any type of agent formula requires a certain constant amount of work (e.g. labelling) which is denoted by c. It is easy to see that, in the best case, evaluation is linear in the length of the trace, i.e. $A(\phi) \in \Omega(t)$. In any other case, each state within the entire trace of length t needs to be labelled with the evaluation result for each subformula of ϕ. The evaluation of each subformula adds $t \cdot c$ to the overall time complexity. We thus have $A(\phi) \in O(t \cdot |\phi|)$.

Let us now turn to the complexity of evaluating full simLTL formulae. The evaluation is not only dependent upon the length of the underlying trace, but also on the number of agents (denoted n) in the population. Let $\Psi_e \subset \Psi$ denote the set of *elementary formulae*, i.e. formulae which do not contain quantifiers or selection operators. The recurrence relation for evaluating any $\psi_e \in \Psi_e$ on groups of n agents and t time steps, denoted $G(\psi_e, n, t)$, can then be given as follows:

$$G(pred, n, t) = t \cdot c \tag{6.9}$$

$$G(\phi, n, t) = A(\phi, t) \tag{6.10}$$

$$G(\neg\, \psi, n, t) = t \cdot c + G(\psi, n, t) \tag{6.11}$$

$$G(\psi_1 \wedge \psi_2, n, t) = t \cdot c + G(\psi_1, n, t) + G(\psi_2, n, t) \tag{6.12}$$

$$G(\psi_1 \vee \psi_2, n, t) = t \cdot c + G(\psi_1, n, t) + G(\psi_2, n, t) \tag{6.13}$$

$$G(aval_1 \trianglelefteq aval_2, n, t) = t \cdot c \tag{6.14}$$

$$G(\mathbf{X}\ \psi, n, t) = t \cdot c + G(\psi, n, t) \tag{6.15}$$

$$G(\psi_1\ \mathbf{U}\ \psi_2, n, t) = t \cdot c + G(\psi_1, n, t) + G(\psi_2, n, t) \tag{6.16}$$

$$G(\psi_1\ \mathbf{R}\ \psi_2, n, t) = t \cdot c + G(\psi_1, n, t) + G(\psi_2, n, t) \tag{6.17}$$

In best case, i.e. if the parse tree of an elementary formula $\psi_e \in \Psi_e$ is of size 1 and contains just a single element, we only need a constant number of operations to perform the evaluation once for each state in the trace. We thus have $G(\psi_e) \in \Omega(t)$. Otherwise, if the parse tree is of size > 1, the evaluation of each subformula of ψ_e adds a constant element to the overall complexity, i.e. we have $G(\psi_e, n) = t \cdot |\psi_e|$. We can thus conclude that $G(\psi_e) \in O(t \cdot |\psi_e|)$.

Let us now look at non-elementary formulae $\psi \in \Psi \setminus \Psi_e$, i.e. formulae containing quantifiers and the selection operator. The recurrence relation can be given as follows:

$$G(\forall\, \psi, n, t) = t \cdot c + n \cdot G(\psi, n, t) \tag{6.18}$$

$$G(\exists^{\trianglelefteq\, k}\ \psi, n, t) = t \cdot c + n \cdot G(\psi, n, t) \tag{6.19}$$

$$G(\exists_r^{\trianglelefteq\, k}\ \psi, n, t) = t \cdot c + n \cdot G(\psi, n, t) \tag{6.20}$$

$$\begin{aligned} G(\langle\!\langle \phi \rangle\!\rangle\ \psi, n, t) &= t \cdot c + n \cdot G(\phi, n, t) + n \cdot G(\psi, n, t) \\ &= t \cdot c + n \cdot A(\phi, t) + n \cdot G(\psi, n, t) \\ &= t \cdot c + n \cdot (A(\phi, t) + G(\psi, n, t)) \end{aligned} \tag{6.21}$$

In best case, the inner formula (i.e. the formula nested within a quantifier or the selection operator) is elementary and only contains a single constant time operation, in which case $G(\psi) = t \cdot n \cdot c$. We can thus conclude that $G(\psi) \in \Omega(t \cdot n)$. In all other cases, the evaluation of each non-elementary

subformula adds a factor of $t + n$ to the total evaluation time. For m nested non-elementary formulae, we thus get a time complexity which is in $O((t + n)^m)$. In worst case, formula ψ consists of nested quantified formulae almost exclusively, e.g. $\forall\forall\forall ...\phi$. In this case, all n agents need to be looped over for each element in the parse tree of ψ. Let $\forall^m \varphi$ denote a formula with m nested universal quantifiers and $\varphi \in \Psi_e$. Based on Equation 6.18, the complexity of evaluating this formula can then be defined as follows:

$$\begin{aligned} G(\forall^m \varphi, n, t) &= t \cdot c + n \cdot (t \cdot c + n \cdot (... \cdot (t \cdot c + n \cdot G(\varphi, n, t)))...) \\ &= t \cdot c + n \cdot t \cdot c + n^2 \cdot t \cdot c + n^3 \cdot t \cdot c + ... + n^m \cdot G(\varphi, n, t) \end{aligned} \tag{6.22}$$

For any $\psi \in \Psi$, we thus have $G(\psi) \in O(t \cdot n^{|\psi|})$, i.e., in worst case, evaluation is polynomial in the number of agents and exponential in the size of the formula.

It is important to note, however, that the worst case in which a formula only contains nested universal quantifiers is highly unlikely to ever occur. In a realistic scenario, it is reasonable to assume that the number of nested quantifiers and selected statements is very small, typically between 1 and 3. It is thus fair to assume the existence of a constant b which defines the upper bound for the *nesting factor* of a formula ψ. In this case we get a time complexity of $O(t \cdot n^b)$ which is still polynomial in the number of agents but no longer exponential in the size of the formula. An example of a property with nesting factor 2 was given in Section 5.4.1 (Formula 5.47).

6.3 An online monitoring algorithm

The exhaustive evaluation algorithm described in the previous section can be applied to any existing, well-formed simulation trace. It checks the validity of a formula ψ based on a recursive labelling function which iteratively traverses the trace in order to label each state with those subformulae ψ_i of ψ which hold in the respective state. It is exhaustive since it requires the labelling of all states with all subformulae before the validity of the entire formula can be determined.

Due to its exhaustive nature, it does not offer any advantage for formulae which can be either satisfied or refuted early, i.e. before the final state has been reached. Given that, in order to explore the space to a sufficient degree, a high number of individual traces need to be examined, an exhaustive approach

is clearly suboptimal and only suitable for small systems or projects which are not time-critical. Furthermore, due to its reliance upon repeated traversal of the same trace (once for each subformula), the exhaustive algorithm is not usable as a monitor in a runtime verification context.

In this section, we present an online evaluation algorithm which aims to solve the problems of the naïve version presented above. It is 'online' because it is capable of checking a running system on-the-fly, thereby reporting any problem as soon as it arises. When used as a monitor in a real-time setting, this is clearly beneficial: any violation reported by the checker can be used to stop the simulation immediately, thus avoiding unnecessary computation and speeding up space exploration. But even if performed on a historical set of simulation output traces, an online approach is advantageous: since errors are reported as early as possible, the evaluation of properties which can be refuted early (e.g. invariants) or satisfied early (e.g. reachability properties) is very quick. On average, we get a significant speedup over the exhaustive approach.

The idea of runtime verification is not new and has been discussed extensively in the literature (see Section 2.4.4). The approach described in this work is influenced by Pnueli's work on *temporal testers* [202]. Briefly summarised, a temporal tester can be described as a monitoring process which observes a trace against the background of given correctness criteria (e.g. formulated in LTL) and reports for each state in the trace whether the given formula has been satisfied, violated, or not decided yet. A temporal tester is called optimal if it is able to extract as much information from the trace as possible and is thus able to report any definite results as soon as possible. Temporal testers have been implemented for a variety of temporal logics; for the remainder of this work, we focus on the development of a temporal tester approach for simLTL.

Temporal testers are realised by exploiting *expansion laws* that can be associated with the basic temporal operators. The expansion laws that hold for conventional LTL were introduced in Equations 5.1 and 5.2 in Section 5.2.2. They are briefly recalled below:

$$\phi_1 \,\mathbf{U}\, \phi_2 \equiv \phi_2 \vee (\phi_1 \wedge \mathbf{X}(\phi_1 \,\mathbf{U}\, \phi_2))$$
$$\phi_1 \,\mathbf{R}\, \phi_2 \equiv \phi_2 \wedge (\phi_1 \vee \mathbf{X}(\phi_1 \,\mathbf{R}\, \phi_2)$$

Intuitively, an expansion law splits up the given formula into two parts: an *immediate part* which needs to be satisfied in the current state and a *future part* which needs to be satisfied by the remaining trace in

order for the given formula to be satisfied. It is useful to think about these two parts as *immediate* and *future obligations*.

As described in Section 5.2.2, simLTL incorporates the finite trace semantics of FLTL [173]; the existence of a weak and a strong version of 'next' which realise both the 'universal' and the 'existential' view on this operator invalidate those expansion laws. As a consequence, they had to be reformulated as follows:

$$\phi_1 \,\mathbf{U}\, \phi_2 \equiv \phi_2 \vee (\phi_1 \wedge \mathbf{X}_{\exists}(\phi_1 \,\mathbf{U}\, \phi_2))$$
$$\phi_1 \,\mathbf{R}\, \phi_2 \equiv \phi_2 \wedge (\phi_1 \vee \mathbf{X}_{\forall}(\phi_1 \,\mathbf{R}\, \phi_2))$$

These expansion laws formulate the basis for the runtime evaluation of arbitrary simLTL formulae, as described in the following sections. An important first step which simplifies the subsequent evaluation significantly is to convert any formula into *Positive Normal Form (PNF)*. Fortunately, as described by Bauer *et al.*, FLTL satisfies all fundamental LTL equivalence laws and any FLTL formula can therefore be translated into PNF [24]. Since simLTL follows the semantics of FLTL, this is also true of simLTL. The translation is described in the next section.

6.3.1 Translation into Positive Normal Form (PNF)

The final preprocessing prior to the actual evaluation is the translation of the respective simLTL formula into PNF, a canonical and easier to manipulate representation in which only a basic set of operators are allowed and all negations are 'pushed inwards' (i.e. may only appear in front of predicates) [12]. In general, any LTL formula can be translated into PNF. The translation is important for two reasons. As described further below, the online algorithm may produce tuples of *immediate* and *future obligations* for each subformula and for each state during the evaluation. Allowing negations to appear in front of obligations would complicate the evaluation. Furthermore, by reducing the semantics to a core set of operators, the evaluation algorithm needs to distinguish fewer cases which facilitates their implementation.

A function which translates arbitrary simLTL agent formulae into their PNF is shown in Algorithm 5. The equivalent function for full simLTL formulae is shown in Algorithm 6. In addition to the LTL

[1] We use '$\overline{\trianglelefteq}$' to denote the complementary operator of $\trianglelefteq$ such that, for example, $\overline{<} = \geq$, $\overline{\leq} = >$, etc.

Algorithm 5 Outline of function $\texttt{toPNF}_a$ for translating agent formulae into PNF

Require: simLTL agent formula ϕ

if $\phi = \mathit{pred}$ **then return** pred
if $\phi = \neg\, \phi_1$ **then**
  if $\phi_1 = \mathit{pred}$ **then return** $\neg\, \mathit{pred}$
  if $\phi_1 = (\rho_1 \wedge \rho_2)$ **then return** $\texttt{toPNF}_a(\neg\, \rho_1 \vee \neg\, \rho_2)$
  if $\phi_1 = (\rho_1 \vee \rho_2)$ **then return** $\texttt{toPNF}_a(\neg\, \rho_1 \wedge \neg\, \rho_2)$
  if $\phi_1 = (\mathit{aval}_1 \trianglelefteq \mathit{aval}_2)$ **then return** $(\mathit{aval}_1 \overline{\trianglelefteq}^1 \mathit{aval}_2)$
  if $\phi_1 = \mathbf{X}_\forall\, \rho$ **then return** $(\mathbf{X}_\exists\, \texttt{toPNF}_a(\neg\, \rho))$
  if $\phi_1 = \mathbf{X}_\exists\, \rho$ **then return** $(\mathbf{X}_\forall\, \texttt{toPNF}_a(\neg\, \rho))$
  if $\phi_1 = (\rho_1\, \mathbf{U}\, \rho_2)$ **then return** $\texttt{toPNF}_a(\neg\, \rho_1\, \mathbf{R}\, \neg\, \rho_2)$
  if $\phi_1 = (\rho_1\, \mathbf{R}\, \rho_2)$ **then return** $\texttt{toPNF}_a(\neg\, \rho_1\, \mathbf{U}\, \neg\, \rho_2)$
end if
if $\phi = (\phi_1 \wedge \phi_2)$ **then return** $(\texttt{toPNF}_a(\phi_1) \wedge \texttt{toPNF}_a(\phi_2))$
if $\phi = (\phi_1 \vee \phi_2)$ **then return** $(\texttt{toPNF}_a(\phi_1) \vee \texttt{toPNF}_a(\phi_2))$
if $\phi = (\mathit{aval}_1 \trianglelefteq \mathit{aval}_2)$ **then return** $(\mathit{aval}_1 \trianglelefteq \mathit{aval}_2)$
if $\phi = \mathbf{X}_\forall\, \phi_1$ **then return** $(\mathbf{X}_\forall\, \texttt{toPNF}_a(\phi_1))$
if $\phi = \mathbf{X}_\exists\, \phi_1$ **then return** $(\mathbf{X}_\exists\, \texttt{toPNF}_a(\phi_1))$
if $\phi = (\phi_1\, \mathbf{U}\, \phi_2)$ **then return** $((\texttt{toPNF}_a(\phi_1)\, \mathbf{U}\, (\texttt{toPNF}_a(\phi_2)))$
if $\phi = (\phi_1\, \mathbf{U}\, \phi_2)$ **then return** $((\texttt{toPNF}_a(\phi_1)\, \mathbf{R}\, (\texttt{toPNF}_a(\phi_2)))$

Algorithm 6 Outline of function $\texttt{toPNF}_g$ for translating full simLTL formulae into PNF

Require: simLTL formula ψ

if $\psi = \mathit{pred}$ **then return** pred
if $\psi = \neg\, \psi_1$ **then**
  if $\psi_1 = \mathit{pred}$ **then return** $\neg\, \mathit{pred}$
  if $\psi_1 = (\rho_1 \wedge \rho_2)$ **then return** $\texttt{toPNF}_g(\neg\, \rho_1 \vee \neg\, \rho_2)$
  if $\psi_1 = (\rho_1 \vee \rho_2)$ **then return** $\texttt{toPNF}_g(\neg\, \rho_1 \wedge \neg\, \rho_2)$
  if $\psi_1 = (\mathit{gval}_1 \trianglelefteq \mathit{gval}_2)$ **then return** $(\mathit{gval}_1 \overline{\trianglelefteq}\ \mathit{gval}_2)$
  if $\psi_1 = \mathbf{X}_\forall\, \rho$ **then return** $(\mathbf{X}_\exists\, \texttt{toPNF}_g(\neg\, \rho))$
  if $\psi_1 = \mathbf{X}_\exists\, \rho$ **then return** $(\mathbf{X}_\forall\, \texttt{toPNF}_g(\neg\, \rho))$
  if $\psi_1 = (\rho_1\, \mathbf{U}\, \rho_2)$ **then return** $\texttt{toPNF}_g(\neg\, \rho_1\, \mathbf{R}\, \rho_2)$
  if $\psi_1 = (\rho_1\, \mathbf{R}\, \rho_2)$ **then return** $\texttt{toPNF}_g(\neg\, \rho_1\, \mathbf{U}\, \neg\, \rho_2)$
  if $\psi_1 = \langle\!\langle \phi \rangle\!\rangle\, \rho$ **then return** $\langle\!\langle \phi \rangle\!\rangle(\texttt{toPNF}_g(\neg\, \rho))$
  if $\psi_1 = \forall\, \rho$ **then return** $\exists^{\geq 1}(\texttt{toPNF}_g(\neg\, \rho))$
  if $\psi_1 = \exists^{\trianglelefteq k}\, \rho$ **then return** $\exists^{\neg\ (\trianglelefteq k)}(\texttt{toPNF}_g(\neg\, \rho))$
end if
if $\psi = (\psi_1 \wedge \psi_2)$ **then return** $(\texttt{toPNF}_g(\psi_1) \wedge \texttt{toPNF}_g(\psi_2))$
if $\psi = (\psi_1 \vee \psi_2)$ **then return** $(\texttt{toPNF}_g(\psi_1) \vee \texttt{toPNF}_g(\psi_2))$
if $\psi = (\mathit{gval}_1 \trianglelefteq \mathit{gval}_2)$ **then return** $(\mathit{gval}_1 \trianglelefteq \mathit{gval}_2)$
if $\psi = \mathbf{X}_\forall\, \phi_1$ **then return** $(\mathbf{X}_\forall\, \texttt{toPNF}_g(\psi_1))$
if $\psi = \mathbf{X}_\exists\, \phi_1$ **then return** $(\mathbf{X}_\exists\, \texttt{toPNF}_g(\psi_1))$
if $\psi = (\psi_1\, \mathbf{U}\, \psi_2)$ **then return** $(\texttt{toPNF}_g(\psi_1)\, \mathbf{U}\, (\texttt{toPNF}_g(\psi_2))$
if $\psi = (\psi_1\, \mathbf{R}\, \psi_2)$ **then return** $(\texttt{toPNF}_g(\psi_1)\, \mathbf{R}\, (\texttt{toPNF}_g(\psi_2))$
if $\psi = \langle\!\langle \phi \rangle\!\rangle\, \psi_1$ **then return** $\langle\!\langle \phi \rangle\!\rangle(\texttt{toPNF}_g(\psi_1))$
if $\psi = \forall\, \psi_1$ **then return** $\forall(\texttt{toPNF}_g(\psi_1))$
if $\psi = \exists^{\trianglelefteq k}\, \psi_1$ **then return** $\exists^{\trianglelefteq k}(\texttt{toPNF}_g(\psi_1))$

fragment of simLTL, Algorithm 6 makes use of the logic for negating selection statements as well as universal and existential quantifiers (see Sections 5.4.1 and 5.4.2) in order to push the negation inwards.

6.3.2 Evaluating simLTL agent formulae

Given the PNF version of a simLTL formula, its evaluation can now take place. According to its design in an online fashion, the evaluation of simLTL formulae starts with the initial state of a simulation trace and stops once (i) a violation has been detected, or (ii) the formula is satisfied. The evaluation of a formula on a trace consists of two steps: the evaluation of the immediate obligation and the (optional) creation of a future obligation. According to the hierarchical structure of simLTL formulae, the algorithm described below is also hierarchical.

We start the description with the innermost and most fundamental part — the evaluation of agent formulae on agent states. Before this can be done, however, a number of data structures need to be introduced. We first need to define the structure of an obligation. On an agent level, an obligation is simply defined as a formula that an agent promises to satisfy. Agent obligations are thus equivalent to the set of agent formulae Φ together with an atom $none$ which denotes the absence of an obligation. In order to clarify subsequent descriptions, we introduce the following alias name:

$$AObligation == \Phi \cup \{none\} \tag{6.23}$$

We further define an *agent evaluation result* as a tuple comprising the result of the immediate obligation and a (not yet evaluated) future obligation ϕ':

$$AResult == Boolean \times AObligation \tag{6.24}$$

The evaluation of agent formulae is performed by a recursive function $\texttt{Check}_a$ shown in Algorithm 7. It accepts an agent formula as input, evaluates it recursively on the state and returns an instance of type $AResult$. The evaluation may produce three different results:

1. $\texttt{Check}_a(\phi)$ returns (true, $none$): In this case, formula ϕ has been satisfied
2. $\texttt{Check}_a(\phi)$ returns (false, ϕ'): In this case, formula ϕ has been refuted
3. $\texttt{Check}_a(\phi)$ returns (true, ϕ'): In this case, formula ϕ has neither been satisfied nor refuted yet

Algorithm 7 Outline of function $\texttt{Check}_a$ for evaluating an agent formula on an agent state

Require: simLTL agent formula ϕ, agent state s

1: **if** $\phi = (pred, X)$ **then return** $(P_a(pred, s, X), none)$
2: **if** $\phi = \neg\, \phi_1$ **then return** $\neg\, \texttt{Check}_a(\phi_1)$
3: **if** $\phi = (\phi_1 \wedge \phi_2)$ **then return** $\texttt{Check}_a(\phi_1)\ \&\&_a\ \texttt{Check}_a(\phi_2)$
4: **if** $\phi = (\phi_1 \vee \phi_2)$ **then return** $\texttt{Check}_a(\phi_1)\ ||_a\ \texttt{Check}_a(\phi_2)$
5: **if** $\phi = (aval_1 \trianglelefteq aval_2)$ **then return** $(\texttt{Eval}_a(aval_1) \trianglelefteq \texttt{Eval}_a(aval_2), none)$
6: **if** $\phi = \mathbf{X}_\forall \phi_1$ **then return** $(\texttt{finalState?}, none)\ ||_a\ (\text{true}, \phi_1)$
7: **if** $\phi = \mathbf{X}_\exists \phi_1$ **then return** $(\neg\, \texttt{finalState?}, \phi_1)$
8: **if** $\phi = (\phi_1\ \mathbf{U}\ \phi_2)$ **then return** $\texttt{Check}_a(\phi_2)\ ||_a\ (\texttt{Check}_a(\phi_1)\ \&\&_a\ (\text{true}, \phi_1\ \mathbf{U}\ \phi_2))$
9: **if** $\phi = (\phi_1\ \mathbf{R}\ \phi_2)$ **then return** $\texttt{Check}_a(\phi_2)\ \&\&_a\ (\texttt{Check}_a(\phi_1)\ ||_a\ (\text{true}, \phi_1\ \mathbf{R}\ \phi_2))$

Algorithm 8 Outline of function $\&\&_a$ for building the conjunction of two agent evaluation results

Require: Evaluation result tuples r_1 and r_2

1: **if** r_1 is (true, ϕ_1) and r_2 is (true, ϕ_2) **then return** $(\text{true}, \phi_1 \wedge \phi_2)$
2: **else return** $(\text{false}, \text{true})$

Algorithm 9 Outline of function $||_a$ for building the conjunction of two agent evaluation results

Require: Evaluation result tuples r_1 and r_2

1: **if** $r_1 = (\text{false}, \phi_1)$ and $r_2 = (\text{false}, \phi_2)$ **then return** $(\text{false}, \text{true})$
2: **if** $r_1 = (\text{false}, \phi_1)$ and $r_2 = (\text{true}, \phi_2)$ **then return** (true, ϕ_2)
3: **if** $r_1 = (\text{true}\, \phi_1)$ and $r_2 = (\text{false}, \phi_2)$ **then return** (true, ϕ_1)
4: **if** $r_1 = (\text{true}\, \phi_1)$ and $r_2 = (\text{true}, \phi_2)$ **then return** $(\text{true}, \phi_1 \vee \phi_2)$

The algorithm makes use of two other binary functions, $\&\&_a$ and $||_a$, which are used as infix operators. Each of them accepts two evaluation result tuples as arguments and builds their conjunction or disjunction, respectively. The return value of either function call is another evaluation result tuple. The logic of $\&\&_a$ is shown in Algorithm 8, the logic of $||_a$ is shown in Algorithm 9. `finalState?` denotes a function which returns true if the state that the formula is being evaluated on is the final one within the current trace. Finally, as described in the previous chapter, predicates are interpreted as calls to Boolean functions. This is indicated by the use of the agent predicate function in Line 1 of Algorithm 7.

6.3.3 Evaluating simLTL formulae

We can now shift the focus to full simLTL formulae whose evaluation is more intricate. Similar to the agent case, we start the description by introducing a recursive algebraic data type $GObligation$ which describes the structure of a *global* or *group obligation*[2].

[2]Note that $c\langle\langle U \rangle\rangle$ describes a *free type* in Z (see Appendix A) and is thus not to be confused with the similar sequence notation.

$$
\begin{aligned}
GObligation ::= \; & AGO\ \langle\!\langle \Psi \times \mathbb{F}\, Ag \rangle\!\rangle \\
| \; & IGO\ \langle\!\langle \mathbb{F}(GObligation \times Ag) \times (CompOp \times \mathbb{N}) \rangle\!\rangle \\
| \; & Disj\ \langle\!\langle GObligation \times GObligation \rangle\!\rangle \\
| \; & Conj\ \langle\!\langle GObligation \times GObligation \rangle\!\rangle \\
| \; & none
\end{aligned}
$$

Before describing the constructors in more detail, it is useful to discuss the concept of a future obligation for a full simLTL formula in more detail. As described above, a future obligation represents a promise that needs to be fulfilled in the next step in order for the formula to be satisfied. Since obligations on a group level concern *groups* of multiple individual agents, several possible cases may arise:

1. The current group *as a whole* needs to satisfy an obligation in the next step
2. Several agents within the current group need to satisfy an obligation *as a group* in the next step
3. Several agents within the current group need to satisfy obligations *individually* in the next step
4. A logical conjunction or disjunction of cases 1-3 needs to be satisfied in the next step
5. No future obligation exists

The first constructor, $AGO\ \langle\!\langle \Psi \times \mathbb{F}\, Ag \rangle\!\rangle$, describes an *aggregate group obligation*, i.e. an obligation which needs to be satisfied *collectively* by a certain group of agents; it accommodates the first two cases in the list above. Its first argument is a full simLTL formula $\psi \in \Psi$, the second argument is a finite set of agents for which the formula must be satisfied. This type of obligation may, for example, result from the evaluation of an aggregate group property.

The second constructor, $IGO\ \langle\!\langle \mathbb{F}(GObligation \times Ag) \times (CompOp \times \mathbb{N}) \rangle\!\rangle$, represents an *individual group obligation* and accommodates the third case in the list above. Here, a set of obligations is to be satisfied *individually* for different agent substitutions. The obligation accepts a finite set of tuples, each of which contains a group obligation as well as an agent; it also contains a *satisfaction constraint* in the form of a tuple comprising a comparison operator drawn from the set $CompOp == \{<, >, \leq, \geq, =, \neq\}$ and a natural number which describes the number of agents within the group that need to satisfy the given criterion. The evaluation of this type of obligation can be seen as an iteration over the given group of agents, for each of which the given sub-obligation is evaluated. This type of obligation may, for example, result from the evaluation of a universally quantified simLTL formula.

Algorithm 10 Outline of function $\texttt{Check}_g$ for evaluating a full simLTL formula on a group state

Require: simLTL formula ψ, group state s, agent in scope a (default = $rand$)

1: **if** $\psi = (pred, X)$ **then return** $(P_g(pred, s, X), none)$
2: **if** $\psi = \phi$ **then return** $\texttt{CheckAgentFormula}(\phi, s, a)$
3: **if** $\psi = \neg\, \psi_1$ **then return** $\neg\, \texttt{Check}_g(\psi_1, s, a)$
4: **if** $\psi = (\psi_1 \wedge \psi_2)$ **then return** $\texttt{Check}_g(\psi_1, s, a)\ \&\&_g\ \texttt{Check}_g(\psi_2, s, a)$
5: **if** $\psi = (\psi_1 \vee \psi_2)$ **then return** $\texttt{Check}_g(\psi_1, s, a)\ ||_g\ \texttt{Check}_g(\psi_2, s, a)$
6: **if** $\psi = (aval_1 \trianglelefteq aval_2)$ **then return** $(\texttt{EVAL}_g(aval_1) \trianglelefteq \texttt{EVAL}_g(aval_2), none)$
7: **if** $\psi = \mathbf{X}_\forall\, \psi_1$ **then return** $(\texttt{finalState?}, none) \vee (\text{true}, AGO(\psi_1\ \text{dom}\ gs))$
8: **if** $\psi = \mathbf{X}_\exists\, \psi_1$ **then return** $(\neg\, \texttt{finalState?}, AGO(\psi_1\ \text{dom}\ gs))$
9: **if** $\psi = (\psi_1\ \mathbf{U}\ \psi_2)$ **then return**
$\texttt{Check}_g(\psi_2, s, a)\ ||_g\ (\texttt{Check}_g(\psi_1, s, a)\ \&\&_g\ \texttt{Check}_g(\mathbf{X}_\exists\, \psi), s, a))$
10: **if** $\psi = (\psi_1\ \mathbf{R}\ \psi_2)$ **then return**
$\texttt{Check}_g(\psi_2, s, a)\ \&\&_g\ (\texttt{Check}_g(\psi_2, s, a)\ ||_g\ \texttt{Check}_g(\mathbf{X}_\forall\, \psi), s, a))$
11: **if** $\psi = \langle\!\langle \phi \rangle\!\rangle\, \psi_1$ **then return** $\texttt{CheckSelection}(\phi, \psi_1, s, a)$
12: **if** $\psi = \forall\, \psi_1$ **then return** $\texttt{CheckForAll}(\psi_1, s, a)$
13: **if** $\psi = \exists^{\trianglelefteq\, k}\, \psi_1$ **then return** $\texttt{CheckExistN}(\psi_1, s, a)$
14: **if** $\psi = \exists_r^{\trianglelefteq\, k}\, \psi_1$ **then return** $\texttt{CheckExistP}(\psi_1, s, a)$

Constructors 3 and 4, $Disj\ \langle\!\langle GObligation \times GObligation \rangle\!\rangle$ and $Conj\ \langle\!\langle GObligation \times GObligation \rangle\!\rangle$, respectively, allow for the recursive definition of conjunctions or disjunctions of agent obligations and thus accommodate the fourth case in the list above.

Finally, constructor 5 denotes an empty obligation; this is important for cases where no future obligation exists (e.g. in the case of non-temporal formulae).

Analogous to the individual case, a *group evaluation result* is defined as a tuple with a Boolean value representing the result of the immediate obligation and the (not yet evaluated) future obligation:

$$GResult == Boolean \times GObligation \tag{6.25}$$

This brings us to the description of the evaluation algorithms for full simLTL formulae and obligations. Similar to the individual case, the most elementary function is $\texttt{Check}_g$ which evaluates a simLTL formula on a group state (see Algorithm 10). The evaluation of unquantified formulae (Lines 1–10) is largely similar to the one for agent formulae (except for Line 2 which is described below). It is important to note that, in the case of $\mathbf{X}_\forall$ and $\mathbf{X}_\exists$ (Lines 7 and 8), an obligation of type AGO is created. It restricts the set of agents which need to satisfy the future obligation to those which are members of the current group state. The evaluation of an individual agent formula (Line 2) as well as those of full formulae including selection or quantification (Lines 11–14) are described below. Since the evaluation of full

simLTL formulae returns different types of future obligations than agent formulae, the binary operators also need to be redefined. This is done further below.

Algorithm 11 Outline of function `CheckAgentFormula`

Require: Agent formula ϕ, group state gs, agent in scope a (default = $rand$)
1: **if** $a = rand$ **then** $a \sim U(\text{dom } gs)$
2: $chk := \texttt{Check}_a(gs[a], \phi)$
3: **if** $(chk.1 = \text{true}) \wedge (chk.2 = none)$ **then**
4: **return** $(\text{true}, none)$
5: **else if** $chk.1 = \text{true}$ **then**
6: **return** $(\text{true}, AGO(chk.2\ \{a\}))$
7: **end if**
8: **return** $(\text{false}, none)$

Function `CheckAgentFormula`: The first function to be discussed is `CheckAgentFormula` (see Algorithm 11) which evaluates an agent formula on a group state. As described in Section 5.4, each agent formula nested within a simLTL formula needs to be indexed, i.e. needs to be accompanied by a variable which is substituted with the agent that the formula is to be evaluated upon. If the variable is free at the time of evaluation, i.e. no substitution has yet been made (denoted by the default value $rand$), then an agent from the current group state is selected randomly by drawing from a uniform distribution (as described in Line 1). This random selection achieves the desired sampling effect from the set of agent states formally described in Section 3.6. The agent formula is then checked on the selected agent state using function $\texttt{Check}_a$ described in Section 6.3.2 above; it returns a tuple with the result of the immediate obligation and the (not yet evaluated) future obligation. If the immediate obligation is satisfied and no future obligation has been produced, the entire formula is also satisfied (see Line 4). If the immediate solution is satisfied but a future obligation has been created, a new obligation of type AGO which only applies to the agent currently in scope (a) is created (see Line 6). In all other cases, the formula is refuted (see Line 8).

Algorithm 12 Outline of function `CheckSelection`

Require: Atemporal agent formula ϕ, group state gs, simLTL formula ψ, agent in scope a
1: $gs' := \{ag : Ag \mapsto AState \mid ag \in gs \wedge \texttt{Check}_a(ag.2, \phi).1 = \text{true}\}$
2: **return** $\texttt{Check}_g(\psi, gs', rand)$

Function `CheckSelection`: As the name of function `CheckSelection` (see Algorithm 12) suggests, it evaluates a simLTL formula ψ which is enclosed in a selection statement on a given group

[2] *first* and *second* refers to the first and second element of a tuple, respectively.

state. The selection statement itself contains an agent formula ϕ and the first step is to determine those agents which satisfy ϕ. This is done in the first line of the algorithm where a restricted group state gs' is created; intuitively, gs' comprises only the states of those agents which satisfy property ϕ. This group state is then passed to function $\texttt{Check}_g$, together with simLTL formula ψ and the undefined agent in scope. Note that the latter is necessary since the currently valid agent in scope may no longer be a member of the group resulting from the selection. The result of the call to function $\texttt{Check}_g$ is also the result of function `CheckSelection`.

Algorithm 13 Outline of function `CheckForAll`

Require: simLTL formula ψ, group state gs

1: $chk := \{ag : Ag \mid ag \in \text{dom } gs \bullet (\texttt{Check}_g(\psi, gs, ag), ag)\}$
2: $nsat := \{r : (GResult \times Ag) \mid r \in chk \land r.1.1 = \text{false}\}$
3: **if** $nsat \neq \varnothing$ **then**
4: **return** $(\text{false}, none)$
5: **else**
6: $fo := \{r : (GResult \times Ag) \mid r \in chk \land r.1.2 \neq none \bullet (r.1.2, r.2)\}$
7: **if** $fo = \varnothing$ **then return** $(\text{true}, none)$
8: **else return** $(\text{true}, IGO\ (fo, (=, \#fo)))$
9: **end if**

Function `CheckForAll`: The function shown in Algorithm 13 describes the evaluation of a 'forAll' formula. Line 1 creates a set of tuples of type $(GResult \times Ag)$ which contain the results of the evaluation of the enclosed formula for each agent in the group. Remember that the 'forAll' statement does not evaluate an agent formula but a full simLTL formula, the reason for which was described in Section 5.4.1. Because of that, function $\texttt{Check}_g$ is called once for each agent in the current trace in Line 1. Line 2 determines the set of all those agents which do *not satisfy* the formula, i.e. for which the formula has already been violated. If this set is nonempty, i.e. if there is at least one agent in the group for which the evaluation has not been successful (as checked in Line 3), then the entire 'forAll' statement is refuted; otherwise, a future obligation is created. The latter is done by constructing a set of all those non-empty future obligation-agent tuples contained in set chk (see Line 6). If this set is empty, then there are no future obligations and the formula is satisfied. If the set is non-empty, then the future obligations of the individual checks are combined into one overall obligation of type IGO. When evaluating a property on a simulation trace once for each agent within the group, the result may be a set of different obligations that need to be satisfied. For that reason, the resulting future obligation has to be of type $\mathbb{F}(GObligation \times Ag)$, i.e. a set of tuples where the first element is a full simLTL formula (the obligation) and the second element is an agent (the obligee). Since, in the case of 'forAll', *all* agents are

expected to satisfy their obligations in the future, the satisfaction constraint is '$(=, \#fo)$', where $\#fo$ denotes the size of set fo, i.e. the total number of agents in the group.

Algorithm 14 Outline of function `CheckExistN`

Require: simLTL formula ψ, group state gs, comparison operator op, natural number n

1: $chk := \{ag : Ag \mid as \in \text{dom } gs \bullet (\texttt{Check}_g(\psi, gs, ag), ag)\}$
2: $psat := \{r : (GResult \times Ag) \mid r \in chk \land r.1.1 = \text{true}\}$
3: $sat := \{r : (GResult \times Ag) \mid r \in psat \land r.1.2 = none\}$
4: **if** $\#sat\ op\ n$ **then**
5: **return** $(\text{true}, none)$
6: **else**
7: $fo := \{r : (GResult \times Ag) \mid r \in psat \land r.1.2 \neq none \bullet (r.1.2, r.2))\}$
8: **return** $(\text{true}, IGO\ (fo, (op, n - \#sat)))$
9: **end if**

Function `CheckExistN`: The next function to be discussed is `CheckExistN`. It checks whether the number of substitutions of agent identifiers for which the current group satisfies $\psi op\ n$, where $op : CompOp$ is a comparison operator[3] and $n \in \mathbb{N}$. Similar to 'forAll' described above, a check of the enclosed formula is performed once for each agent in the current group and the results are stored in a set chk (see Line 1). Based on that, the set $psat$ of agents which *potentially satisfy* the formula, i.e. those agents for which the formula has not been clearly violated yet, is determined in Line 2. The next step is to determine, within the set $psat$, those agents which *actually satisfy* the formula, i.e. those agents whose immediate obligation result is satisfied and whose future obligation is empty; this is done in Line 3. If the cardinality of the resulting set satisfies the constraints '$op\ n$', then the formula is satisfied; if not, then those future obligations which need to be satisfied in order to satisfy the entire formula are collected and wrapped into an overall obligation of type IGO in Lines 7 and 8. In order to end up with the correct number of agents satisfying the formula, it is important to constrain the future obligations appropriately. This is the purpose of the satisfaction constraint defined in Line 8. Since several agents may have already satisfied their requirements (as determined in Line 3), their number needs to be deducted from the overall number of agents which still need to satisfy the formula.

A description of function `CheckExistP` for evaluating relatively quantified existential statements is omitted here. It is similar to function `CheckExistN` apart from Line 4 in which the *ratio* of agents needs to be calculated instead.

[3] $CompOp == \{<, >, \leq, \geq, =, \neq\}$

6.3.4 Combining group evaluation results and obligations

Algorithm 15 Outline of function $\&\&_g$ for building the conjunction of two group evaluation results

Require: Group evaluation result tuples r_1 and r_2
1: **if** r_1 is (true, ψ_1) and r_2 is (true, ψ_2) **then return** $(\text{true}, \texttt{gOblAnd}(\psi_1, \psi_2))$
2: **else return** $(\text{false}, \mathit{none})$

Algorithm 16 Outline of function $||_g$ for building the conjunction of two group evaluation results

Require: Group evaluation result tuples r_1 and r_2
1: **if** $r_1 = (\text{false}, \psi_1)$ and $r_2 = (\text{false}, \psi_2)$ **then return** $(\text{false}, \mathit{none})$
2: **if** $r_1 = (\text{false}, \psi_1)$ and $r_2 = (\text{true}, \psi_2)$ **then return** (true, ψ_2)
3: **if** $r_1 = (\text{true}, \psi_1)$ and $r_2 = (\text{false}, \psi_2)$ **then return** (true, ψ_1)
4: **if** $r_1 = (\text{true}, \psi_1)$ and $r_2 = (\text{true}, \psi_2)$ **then return** $(\text{true}, \texttt{gOblOr}(\psi_1, \psi_2))$

Algorithm 17 Outline of function `gOblAnd` for building the conjunction of two group obligations

Require: Group obligations σ_1 and σ_2
1: **if** $\sigma_1 = \mathit{AGO}\ (\text{true}, \_)$ and $\sigma_2 = \mathit{AGO}\ (\text{true}, \_)$ **then return** none
2: **if** $\sigma_1 = \mathit{AGO}\ (\text{true}, \varnothing)$ **then return** σ_2
3: **if** $\sigma_2 = \mathit{AGO}\ (\text{true}, \varnothing)$ **then return** σ_1
4: **if** $\sigma_1 = \mathit{AGO}\ (\psi_1, is_1)$ and $\sigma_2 = \mathit{AGO}\ (\psi_2, is_2)$ **then**
5: **if** $is_1 = is_2$ **then return** $\mathit{AGO}\ (\psi_1 \wedge \psi_2, is_1)$
6: **else return** $\mathit{Conj}\ (\sigma_1, \sigma_2)$
7: **end if**
8: **if** $\sigma_1 = \mathit{IGO}\ (\varnothing, \_)$ and $\sigma_2 = \mathit{IGO}\ (\varnothing, \_)$ **then return** none
9: **if** $\sigma_1 = \mathit{IGO}\ (\varnothing, \_)$ **then return** σ_2
10: **if** $\sigma_2 = \mathit{IGO}\ (\varnothing, \_)$ **then return** σ_1
11: **return** $\mathit{Conj}\ (\sigma_1, \sigma_2)$

Algorithm 18 Outline of function `gOblOr` for building the disjunction of two group obligations

Require: Group obligations σ_1 and σ_2
1: **if** $\sigma_1 = \mathit{AGO}\ (\text{true}, \_)$ or $\sigma_2 = \mathit{AGO}\ (\text{true}, \_)$ **then return** none
2: **if** $\sigma_1 = \mathit{AGO}\ (\psi_1, is_1)$ and $\sigma_2 = \mathit{AGO}\ (\psi_2, is_2)$ **then**
3: **if** $is_1 = is_2$ **then return** $\mathit{AGO}\ (\psi_1 \vee \psi_2, is_1)$
4: **else return** $\mathit{Disj}\ (\sigma_1, \sigma_2)$
5: **end if**
6: **if** $\sigma_1 = \mathit{IGO}\ (\varnothing, \_)$ or $\sigma_2 = \mathit{IGO}\ (\varnothing, \_)$ **then return** none
7: **return** $\mathit{Disj}\ (\sigma_1, \sigma_2)$

Similar to the agent case, group evaluation results need to be logically combined in order to form more complex results. Consider again function $\texttt{Check}_g$ described further above (see Algorithm 10). In Line 4, for example, two obligations need to be conjoined; Line 5 shows an example of a disjunction.

The basic logic of the operators is similar to the agent case and shown in Algorithms 15 and 16. Due to the more complicated structure of group obligations, their conjunction and disjunction requires the

treatment of some special cases. For clarity, this logic has been moved to two separate functions, `gOblAnd` and `gOblOr`, which are shown in Algorithms 17 and 18. It is important to note, however, that the algorithms are not yet optimal. They ignore logical relations that may exist between formulae to be combined. For example, when conjoining two future obligations $\mathbf{F}\phi$ and $\mathbf{GF}\phi$ (a case which may result from the evaluation of $\mathbf{GF}\phi$), Algorithm 17 does not recognise that truth of the latter formula implies truth of the former and that evaluation of the former formula can thus be omitted. The algorithm would instead produce a combined obligation of type $\mathit{Conj}\ (\mathbf{G}\phi, \mathbf{GF}\phi)$ which is unnecessarily complex. This may, in certain cases, lead to the creation of highly nested obligations over time which is clearly undesirable.

6.3.5 Checking group obligations

Algorithm 19 Outline of function `CheckGO` for evaluating a group obligation on a group state

Require: SimLTL group obligation obl, group state gs, agent in scope a (default = $rand$)

if $obl = \mathit{AGO}\ (\psi, is)$ **then**
  $gs' := \{ag : \mathit{Ag} \mapsto \mathit{AState} \mid ag \in gs \wedge \texttt{Check}_a(ag.2, \phi).1 = \text{true}\}$
  return $\texttt{Check}_g(\psi, gs', a)$
end if
if $obl = \mathit{IGO}\ xs$ **then return** $\texttt{CheckIGO}(gs, xs)$
if $obl = \mathit{Conj}\ (\psi_1, \psi_2)$ **then**
  $(b_1, o_1) := \texttt{CheckGO}(\psi_1, gs, a)$
  $(b_2, o_2) := \texttt{CheckGO}(\psi_2, gs, a)$
  return $(b_1 \wedge b_2, \texttt{gOblAnd}(o_1, o_2))$
end if
if $obl = \mathit{Disj}\ (\psi_1, \psi_2)$ **then**
  $(b_1, o_1) := \texttt{CheckGO}(gs, \psi_1, a)$
  $(b_2, o_2) := \texttt{CheckGO}(gs, \psi_2, a)$
  return $(b_1 \vee b_2, \texttt{gOblOr}(o_1, o_2))$
end if

We now have all of the functions that we need in order to describe the evaluation of group obligations on group states and, ultimately, on simulation traces. An outline of function `CheckGO` which evaluates a group obligation on a group state is shown in Algorithm 19. It makes use of function $\texttt{Check}_g$ defined above as well as of function `CheckIGO` which is described below.

Since the treatment of individual group obligations is slightly more complicated than that of aggregate group obligations, it has been outsourced to a separate function `CheckIGO` shown in Algorithm 20. The logic of the function becomes clearer when one looks at the type of its second argument:

$$\mathbb{F}(\mathit{GObligation} \times \mathit{Ag}) \times (\mathit{CompOp} \times \mathbb{N})$$

Algorithm 20 Outline of function `CheckIGO`

Require: Group state gs, obligation tuple $xs : \mathbb{F}(GObligation \times Ag) \times (CompOp \times \mathbb{N})$

1: $chk := \{r : (GObligation \times Ag) \mid r \in xs.1 \bullet (\texttt{CheckGO}(r.1, gs, r.2), r.2)\}$
2: $op := xs.2.1$
3: $card := xs.2.2$
4: $psat := \{r : (GResult \times Ag) \mid r \in chk \land r.1.1 = \text{true}\}$
5: $sat := \{r : (GResult \times Ag) \mid r \in psat \land r.1.2 = none\}$
6: **if** $\#sat\ op\ card$ **then**
7: **return** $(\text{true}, none)$
8: **else**
9: $fo := \{r : (GResult \times Ag) \mid r \in psat \land r.1.2 \neq none \bullet (r.1.2, r.2)\}$
10: **return** $(\text{true}, IGO\ (fo, (op, card - \#sat)))$
11: **end if**

The argument is a tuple comprising a finite set of simLTL formula-agent identifier tuples (the individual obligations as well as the obligees) and a satisfaction constraint (comparison operator and a natural number). Obligations of type IGO are satisfied if and only if the desired number (specified by the satisfaction constraint) of sub-obligations (specified by the finite set) is satisfied. The evaluation of that particular type of obligation bears some similarity with the evaluation of existentially quantified formulae described above. The evaluation procedure (shown in Algorithm 20) is therefore very similar to function `CheckExistN` described above. In the first step, a set of evaluation results for each sub-obligation is created (see Line 1). The set of those agents *potentially* satisfying their obligations (see Line 4) as well as the set of those agents *already* satisfying their obligations (see Line 5) is constructed. If the size of the latter set satisfies the given constraints, then the obligation is satisfied; otherwise, a set of future obligations is created and wrapped into an overall obligation of type IGO; similar to `ExistN`, the satisfaction constraint needs to be adjusted with respect to those agents which have already satisfied their obligations.

It remains to discuss the evaluation of group obligations on full simulation traces. Given a simulation trace t_s, the truth of an obligation σ is checked by evaluating it on the *initial* state of t_s, i.e. by checking whether $t_s[0] \models \sigma$. In the case of a non-temporal formula, the evaluation will immediately return a definite result; in the case of a temporal formula, the evaluation may produce a future obligation, i.e. a formula which needs to be satisfied in the next state in order for the whole formula to be satisfied. The evaluation of the future obligation might itself produce a future obligation which needs to be satisfied by the subsequent state, etc. This process continues until the entire obligation σ is either satisfied or refuted.

In order for the algorithm described to be optimal, however, it is important to avoid the creation of a full simulation trace prior to evaluation — after all, it could be satisfied or violated in its initial state. To this end, we make the algorithm *lazy*. Instead of passing a simulation trace to the algorithm, a sequence of time steps to be simulated is passed; the simulation of a tick is then requested whenever necessary.

Algorithm 21 Outline of function `CheckTrace` for lazy evaluation of a group obligation on a simulation trace

Require: Sequence of remaining time steps $ticks : \text{seq}\,\mathbb{N}$, group obligation σ

1: **if** $ticks = \langle\,\rangle$ **then return** false
2: **if** $ticks = \langle t \rangle$ **then**
3: $gs := \texttt{Step}(t)$
4: $res := \texttt{CheckGO}(\sigma, gs, rand)$
5: **if** $res = (\text{true}, none)$ **then return** true
6: **if** $res = (\text{false}, none)$ **then return** false
7: **end if**
8: **if** $ticks = \langle t_1, t_2, ..., t_n \rangle$ **then**
9: $gs := \texttt{Step}(t_1)$
10: $res := \texttt{CheckGO}(\sigma, gs, rand)$
11: **if** $res = (\text{true}, none)$ **then return** true
12: **if** $res = (\text{true}, \sigma')$ **then return** $\texttt{CheckTrace}(\langle t_2, ..., t_n \rangle, \sigma')$
13: **if** $res = (\text{false}, \_)$ **then return** false
14: **end if**

A simple recursive algorithm for this procedure is shown in Algorithm 21. It handles three cases: (i) the underlying simulation trace is empty, (ii) it contains exactly one state ('singleton trace'), and (iii) it contains multiple states. The first case (empty simulation trace) is simple. By definition, an empty simulation trace dissatisfies any formula and the function thus returns false. The third case (simulation trace contains multiple states) is also straightforward. Here, a new group state is requested by calling function `Step` which the time step to be simulated is passed to. The obligation is then evaluated on the returned group state by calling function `CheckGO`. It returns a result tuple which is subject to a subsequent case analysis. The case analysis ensures that the entire evaluation returns a result as soon as the inner formulae have been satisfied. In that way, the optimality of the model checking algorithm with respect to the avoidance of unnecessary computation is guaranteed. If an obligation has been produced, then the function calls itself recursively and evaluates the obligation on the tail of the current trace. The second case (simulation trace contains exactly one state) differs from the third case in a small but important detail. Here, an obligation is expected to be either clearly satisfiable or clearly refutable — the creation of a future obligation is not permitted. The fact that a future obligation will never be created in the final state of a trace is ensured by the semantics of simLTL which are explicitly defined over finite traces; this problem has been discussed extensively in Section 5.2.2.

6.3.6 Time complexity

This section discusses the runtime complexity of the online model checking algorithm described above.

6.3.6.1 Checking individual states

We start the complexity analysis with the case of checking an agent formula ϕ on an agent state and define their time complexity using the following recurrence relation A_s:

$$A_s(pred) = c \tag{6.26}$$

$$A_s(\neg\, \psi) = c + A_s(\psi) \tag{6.27}$$

$$A_s(\psi_1 \wedge \psi_2) = c + A_s(\psi_1) + A_s(\psi_2) \tag{6.28}$$

$$A_s(\psi_1 \vee \psi_2) = c + A_s(\psi_1) + A_s(\psi_2) \tag{6.29}$$

$$A_s(aval_1 \trianglelefteq aval_2) = c \tag{6.30}$$

$$A_s(\mathbf{X}_\forall\, \psi) = c \tag{6.31}$$

$$A_s(\mathbf{X}_\exists\, \psi) = c \tag{6.32}$$

$$A_s(\psi_1\, \mathbf{U}\, \psi_2) = c + A_s(\psi_1) + A_s(\psi_2) \tag{6.33}$$

$$A_s(\psi_1\, \mathbf{R}\, \psi_2) = c + A_s(\psi_1) + A_s(\psi_2) \tag{6.34}$$

Again, c denotes a constant amount of time. We can easily see that $A_s(\phi) \in \Omega(c)$. In worst case, we need to perform a constant time operation once for each node in the parse tree of ϕ. We can thus conclude that $A_s(\phi) \in O(|\phi|)$.

Let us now look at the complexity of evaluating a full simLTL formula on a group state. The recurrence relation G_s for evaluating a formula on a group state of size n can be given as follows:

$$G_s(pred, n) = c \tag{6.35}$$

$$G_s(\phi, n) = A_s(\phi) \tag{6.36}$$

$$G_s(\neg\, \psi, n) = c + G_s(\psi, n) \tag{6.37}$$

$$G_s(\psi_1 \wedge \psi_2, n) = c + G_s(\psi_1, n) + G_s(\psi_2, n) \tag{6.38}$$

$$G_s(\psi_1 \vee \psi_2, n) = c + G_s(\psi_1, n) + G_s(\psi_2, n) \tag{6.39}$$

$$G_s(aval_1 \trianglelefteq aval_2, n) = c \tag{6.40}$$

$$G_s(\mathbf{X}_\forall\, \psi, n) = c \tag{6.41}$$

$$G_s(\mathbf{X}_\exists\, \psi, n) = c \tag{6.42}$$

$$G_s(\psi_1\, \mathbf{U}\, \psi_2, n) = c + G_s(\psi_1, n) + G_s(\psi_2, n) \tag{6.43}$$

$$G_s(\psi_1\, \mathbf{R}\, \psi_2, n) = c + G_s(\psi_1, n) + G_s(\psi_2, n) \tag{6.44}$$

$$G_s(\forall\, \psi, n) = c + n \cdot G_s(\psi, n) \tag{6.45}$$

$$G_s(\exists^{\trianglelefteq\, k}\, \psi, n) = c + n \cdot G_s(\psi, n) \tag{6.46}$$

$$G_s(\exists_r^{\trianglelefteq\, k}\, \psi, n) = c + n \cdot G_s(\psi, n) \tag{6.47}$$

$$\begin{aligned} G_s(\langle\!\langle \phi \rangle\!\rangle\, \psi, n) &= c + n \cdot G_s(\phi, n) + n \cdot G_s(\psi, n) \\ &= c + n \cdot (G_s(\phi, n) + G_s(\psi, n)) \end{aligned} \tag{6.48}$$

In best case, i.e. if the parse tree of an elementary formula ψ contains just one element, we only need a constant number of operations to perform the evaluation. We thus have $G_s(\psi, n) \in \Omega(c)$. In all other cases, the evaluation of each non-elementary subformula adds a factor of n to the total evaluation time. For m nested non-elementary formulae, we thus get a time complexity which is in $O(n^m)$. In worst case, formula ψ exclusively consists of nested quantification or selection operators, as described in Section 6.2.2 above. In this case, all n agents need to be looped over for each element in the parse tree of ψ. We thus have $G_s(\psi, n) \in O(n^{|\psi|})$, i.e. evaluation is polynomial in the number of agents and exponential in the size of the formula. Again, this case is highly unlikely and we can instead assume that there is an upper bound b on the 'nesting level' of a formula, i.e. the number of nested selection or quantification operators. In this case, we get $G_s(\psi, n) \in O(n^b)$.

6.3.6.2 Checking traces

It remains to discuss the time complexity of evaluating full simLTL formulae over entire traces (or fragments thereof). In contrast to the complexity of a single property as discussed above, this involves determining *how many* formulae need to be evaluated throughout the course of an individual simulation run. The recurrence relation G_t for evaluating a simLTL formula with n agents on a trace of length t can be given as follows:

$$G_t(\psi, n, 1) = G_s(\psi, n) \tag{6.49}$$

$$G_t(pred, n, t) = G_s(pred, n) \tag{6.50}$$

$$G_t(\phi, n, t) = G_s(\phi, n) \tag{6.51}$$

$$G_t(\neg\, \psi, n, t) = G_t(\psi, n, t) \tag{6.52}$$

$$G_t(\psi \wedge \psi, n, t) = G_t(\psi, n, t) + G_t(\psi, n, t) \tag{6.53}$$

$$G_t(\psi \vee \psi, n, t) = G_t(\psi, n, t) + G_t(\psi, n, t) \tag{6.54}$$

$$G_t(aval_1 \trianglelefteq aval_2, n, t) = G_s(aval_1 \trianglelefteq aval_2, n) \tag{6.55}$$

$$G_t(\mathbf{X}_\forall\, \psi, n, t) = G_t(\psi, n, t-1) \tag{6.56}$$

$$G_t(\mathbf{X}_\exists\, \psi, n, t) = G_t(\psi, n, t-1) \tag{6.57}$$

$$G_t(\psi_1\, \mathbf{U}\, \psi_2, n, t) = G_t(\psi_2, n, t) + G_t(\psi_1\, \mathbf{U}\, \psi_2, n, t-1) \tag{6.58}$$

$$G_t(\psi_1\, \mathbf{R}\, \psi_2, n, t) = G_t(\psi_2, n, t) + G_t(\psi_1\, \mathbf{R}\, \psi_2, n, t-1) \tag{6.59}$$

$$G_t(\forall\, \psi, n, t) = n \cdot G_t(\psi, n, t) \tag{6.60}$$

$$G_t(\exists^{\trianglelefteq\, k}\, \psi, n, t) = n \cdot G_t(\psi, n, t) \tag{6.61}$$

$$G_t(\exists_r^{\trianglelefteq\, k}\, \psi, n, t) = n \cdot G_t(\psi, n, t) \tag{6.62}$$

$$G_t(\langle\!\langle \phi \rangle\!\rangle\, \psi, n, t) = n \cdot G_t(\phi, n, t) + n \cdot G_t(\psi, n, t) \tag{6.63}$$

We can see that, in the best case, only one formula needs to be evaluated, i.e. $G_t(\psi, n, t) \in \Omega(c)$. In the worst case, in each time step and for each agent in the population, one new formula is created in the case of 'next', 'until' and 'release' and, as a consequence, t formulae need to be evaluated. We thus get $G_t(\psi, n, t) \in O(t \cdot n^{|\psi|})$. It is important to note, however, that this case is highly unlikely to ever occur in reality; it would, for example, require multiple universally quantified formulae which cannot be satisfied or refuted before the final state of a trace to be wrapped into an 'until' or 'release' statement (as illustrated above and in Section 6.2.2). Stating the typical case complexity is difficult because it depends on the nature of the properties formulated. Nevertheless, if we assume that formulae are, on average, decidable after 50% of the total number of ticks have been processed, then we get a complexity of $O\left(\frac{t}{2} \cdot n^b\right)$ where, similar to the discussion of the exhaustive evaluation algorithm in Section 6.2.2, we assume b to be the maximum nesting level of quantifiers and/or selection statements.

This concludes the description of algorithms for the evaluation of simLTL formulae upon individual

traces. The next section extends beyond the border of individual traces and describes an estimation procedure using which the probability of a property being true in the whole state space can be determined.

6.4 Estimating the probability of a property

Determining the *exact* probability of a property being true in a given model requires full exploration of the underlying state space. This is the approach followed by exhaustive probabilistic model checking described in Section 2.4.3.2. Here, a verification result could be "property ϕ is true in 80% of all cases", i.e. property ϕ holds in 80% of all possible simulation traces. As described in Section 2.2.3, determining probabilities is important to understand the internal mechanisms of the model. For most real-world systems, however, exhaustive exploration and therefore exact determination of probabilities is infeasible. Approximate techniques can help to *estimate* probabilities through *sampling*, i.e. by only exploring a certain fragment of the space, and inferring information about the full space. The verification result then changes into something analogous to "the probability that property ϕ is true in 80% of all cases, allowing for an error of $\pm 1\%$, is at least 99%". Determining the probabilities of events is important because it helps modellers to understand better the internal dynamics of a given model. Especially in the presence of randomness, obtaining quantitative results is important and superior over purely qualitative insights. Consider, for example, a complex simulation model in which a property ϕ only holds in one of astronomically many possible traces; in this case, the probability of ϕ is effectively 0. Purely qualitative reachability analysis ('is it possible to reach a certain state, yes or no?'), on the other hand, would return a positive answer and thus assign the same 'level of truth' to ϕ as it would to any other, significantly more frequent, event which is clearly misleading.

It is also important to note, however, that care is advised when probabilities obtained through analysis of a simulation model as part of the verification or internal validation process are to be generalised to the real world. The probabilities obtained through verification are probabilities of events happening *in the model space*; simulation models are always approximate in nature and, as long as the *ontological adequacy* of the model has not been assessed, results produced by the model (including probabilities of certain events) cannot be guaranteed to correspond with their counterparts in the real world. It is the purpose of external validation as described in Section 2.2.3 to analyse the model-phenomenon link and assess the predictive capabilities of the model. We focus on internal validation in this work; when we refer to probabilities of events, we thus always mean their relative occurrence against the background of the model's state space.

Algorithm 22 Sample size calculation procedure `GetNumSamples`

Require: Accuracy ϵ, confidence δ, probability $p = 0.5$
  $\text{cdf}(n) = \sum_{i=0}^{n} \binom{n}{(i*p)-(i*\epsilon)} p^i (1-p)^{n-i}$
  $\mathit{hoeff} := \ln\left(\frac{2}{\delta}\right) \cdot \frac{1}{2\epsilon^2}$
  $L := [1..\mathit{hoeff}]$
  for $i := 1$ to $|L|$ **do**
    $P := \text{cdf}(L[i])$
    if $P \leq \delta$ **then return** $L[i]$
  end for

In the context of model checking, probability estimation through sampling is the idea followed by statistical approaches described in Section 2.4.3.3. The technique used in this work is influenced by the idea of *approximate probabilistic model checking (APMC)* which has first been proposed by Hérault and Lassaigne [117]. The main difference of APMC to other stochastic model checking approaches is that the number of paths that need to be explored in order to achieve a certain level of confidence and accuracy is fixed and independent from the size of the simulation to be verified. This is guaranteed by an application of the *Hoeffding inequality* which provides an upper bound on the number of samples necessary [127] (see Section 2.4.3.3 for details). Other approaches (e.g. those based on hypothesis testing) follow a different approach and adapt the sample size during verification. The estimation procedure proposed by Hérault and Lassaigne, the *Generic Approximation Algorithm (GAA)*, was shown in Algorithm 1 in Section 2.4.3.3.

We use a simple algorithmic procedure to calculate the sample size necessary to achieve a certain level of confidence and accuracy with respect to the verification results. As opposed to a probabilistic *bound* (such as the Hoeffding bound) which is represented by a nice mathematical formula but often overestimates the actually necessary sample size by a significant degree (as exemplified in Figure 6.1 below), the procedure described here is algorithmic but accurate since it operates directly on the Binomial distribution. A similar approach has, for example, been described by Mount [193]. The procedure is shown in Algorithm 22. It uses a function 'cdf' which represents the cumulative distribution function of the Binomial distribution. Given a certain number of trials, each of which has a success probability of p, 'cdf' calculates the probability of a random number X being less than or equal to a given value x. Now, if we want to detect an event with confidence $1-\delta$ and an error of at most $\pm\epsilon$, then we can use 'cdf' to determine the number n of trials necessary to detect an event of probability $\leq \delta+\epsilon$ if the individual trial has a success probability of p. We do not know the value of p, of course, therefore choose $p = 0.5$. For example, let $\epsilon = 0.01$ and $\delta = 0.05$. In a sample of 100 trials, the probability of $(100*0.5)-(100*0.01)$ trials being successful is ≈ 0.46 which is greater than the desired value of 0.05. We can thus conclude

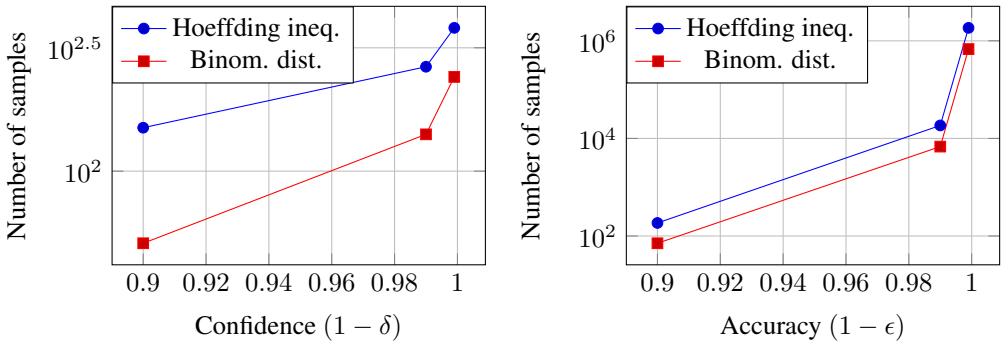


FIGURE 6.1: Relation between sample size and confidence (left) and sample size and accuracy (right) for the Hoeffding bound and the accurate calculation using the Binomial distribution

that a sample size of 100 is not sufficient. According to Algorithm 22, we would need 6,801 sample runs in order to detect the desired event. In contrast, the Hoeffding bound would suggest a much larger sample size of 18,445.

A comparison between the sample size suggested by the Hoeffding bound and that of the algorithmic procedure is shown in Figure 6.1. It becomes apparent that the algorithm does not avoid the quadratic dependency of the sample size in ϵ and thus suffers from the same limitations as the Hoeffding bound. It does, however, return a lower *total sample size* which reduces the number of simulation traces that need to be analysed. In the presence of resource constraints, this can represent a practical advantage. Furthermore, the calculation of the cumulative distribution function can become expensive as n grows. In a real implementation, it is thus advisable to avoid a naïve iteration from 1 to N and perform, for example, a binary search instead.

Algorithm 23 Approximation function `Estimate`

Require: Configuration id id, simLTL formula ψ, confidence δ, accuracy ϵ, fragment size k, number of ticks t

1: $N := \texttt{GetNumSamples}(\epsilon, \delta)$
2: $A := 0$
3: $o := AGO\ \psi\ \{1, 2, ..., n\}$
4: $\texttt{PreConf}(id)$
5: **for** $i := 1$ to N **do**
6: $\quad ticks := \langle 1, 2, ..., t \rangle$
7: $\quad fragments := \texttt{GetFragments}(ticks, k)$
8: $\quad res := \{f : \text{seq}\,\mathbb{N} \mid f \in fragments \bullet \texttt{CheckTrace}(f, o)\}$
9: $\quad A := A + \texttt{SatR}(res)$
10: **end for**
11: $\texttt{PostConf}(id)$
12: **return** A/N

An outline of the overall approximation function `Estimate` is shown in Algorithm 23. As described in Section 3.6, the notion of fragments is important to ensure sampling from the correct distribution. Properties correspond with events and each property can be associated with a maximum fragment length which it requires to be answered definitely. A non-temporal formula requires precisely one state to be answered definitely. As described in Chapter 6, every temporal operator can be reduced to either 'next' or 'until'. A 'next' formula requires exactly two states to be answered definitely, whereas an 'until' formula requires, in worst case, as many states as there are in the trace. In order to determine the ratio of a formula being true on a given trace, we thus need to check it on each trace fragment of length k required as a maximum for a definite evaluation. In case of a temporal formula containing an operator which is derived from 'until', we need as many states as there are in the trace. There is only one sub-trace which satisfies this requirement, namely the same as the original trace. A simulation trace is thus only able to satisfy such a temporal formula once. In the case of the 'next' operator, we need to check the formula on all sub-traces of length 2. In the case of non-temporal formulae, we need to check it on all fragments of length 1, i.e. on all individual states.

In order to realise that functionality, function `Estimate` also accepts an argument k which denotes the fragment length required to answer the property. As described above, in order to be truly online, we avoid producing the entire simulation trace prior to evaluation; instead, a list of ticks which acts as a surrogate for the simulation trace is created. A call to function `GetFragments` returns a list of all trace fragments of *ticks* of given length k which the property is then checked upon by calling `CheckTrace` iteratively. Remember that `CheckTrace` expects a *group obligation* rather than a simLTL formula. To this end, formula ψ is transformed into a group obligation in Line 3[4]. The result of the iterative call to `CheckTrace` is a set of Boolean values which represent the checks of ψ on individual trace fragments. In order to determine the *satisfaction ratio* of ψ on the given set of trace fragments — the ratio of successful versus non-successful checks — we call function `SatR` : $\mathbb{F}\ Boolean \rightarrow \mathbb{R}$ which is straightforward and not further described here.

In order to estimate the probability of ψ, denoted $Pr_k(\psi)$, ψ is evaluated on a sufficiently large sample of simulation trace fragments of size k whose size N is controlled by two parameters, accuracy δ and confidence ϵ, and determined by function `GetNumSamples` described above. The Binomial distribution guarantees that, when averaging the number of successful evaluations over N sample traces, the resulting value deviates from the real probability X by not more than ϵ with probability $1 - \delta$. We thus have the following inequality (irrelevant function arguments are denoted with an asterisk):

[4]Here, n denotes the number of agents in the simulation model

$$Pr(\mid Pr_k(\psi) - \texttt{Estimate}(*, \psi, \delta, \epsilon, k) \mid \leq \epsilon) \geq \delta \tag{6.64}$$

The function also calls two helper functions `PreConf` and `PostConf` which allow for initialisation and finalisation logic related with the current *configuration* to be executed. A configuration is defined as a particular set of initial conditions of the simulation. In order to distinguish between different configurations and react appropriately within the model, a numeric id is passed to function `Estimate`. The notion of configurations will become particularly important in the context of causal analysis described in Section 7.6 further below.

6.5 Summary

This chapter described algorithms for the evaluation of simLTL formulae on simulation traces (or, more precisely, fragments of simulation traces). In Section 6.2, a basic, exhaustive offline algorithm was introduced (see Algorithms 3 and 4). The algorithm requires the existence of a full simulation trace prior to the evaluation of the formula and labels each state in the underlying traces once for each subformula. In cases where an answer to the formula can be found before the final state of the produced trace, computation time is thus wasted unnecessarily.

Section 6.3 continued with the presentation of an optimised algorithm which solves the problems of the exhaustive version by working in an *online fashion* (see Algorithms 7 – 23). By *interleaving simulation and evaluation*, it implements a true *monitor* which returns a result as soon as it can be found. The algorithm is based on the idea of *laziness*: simulation and evaluation are interleaved in such a way that the computation of a new state of the simulation trace is requested only if it is strictly required for evaluation (see Algorithm 21). If the monitor is able to prove satisfaction or violation of a given property at time t, time step $t + 1$ will thus never be computed. By interleaving simulation and evaluation in that way, optimality with respect to *avoiding unnecessary computation* can be achieved: the evaluation algorithm recognises immediately when the truth of a formula has either been clearly proved or disproved and, in the latter case, may cause the entire execution (including simulation) to stop.

The monitoring algorithm focusses on the evaluation of properties on individual traces and serves as the core of the verification framework described in this work. By extending the focus to multiple simulation

traces and their interpretation as sets of *possible worlds* which the simulation is able to generate, the *probability of a property* can be estimated. In order to avoid full exploration of the underlying state space, this is done by *randomly sampling* individual traces through repeated simulation. The number of simulation traces necessary to achieve a certain level of precision with respect to the verification results is determined by a simple algorithmic procedure which operates directly on the Binomial distribution (see Algorithm 22). By varying the sample size, accuracy and confidence can be calibrated as desired.

At this point, it is useful to briefly explore the relationship between the approach chosen in this thesis and symbolic techniques to formal verification [181, 31]. A basic assumption of this work is that the states produced by a simulation are represented *explicitly* in memory, e.g. as sets of key–value pairs. A critical problem of explicit representation in conventional model checking is the exponential growth of the state space which renders the verification of non-trivial systems impossible. However, due to the online nature of the approach described here and the focus on linear temporal logic, only the state currently being analysed needs to be kept in memory and exponentially growing state spaces can be dealt with efficiently. On the other hand, as mentioned in Section 5.1 and described in more detail in Section 10.3.1, it would be interesting to consider the extension of the approach described this thesis to the verification of branching time properties. Depending on the traversal mode, the explicit representation of the state space becomes more problematic. In order to address this problem, states could potentially be represented *symbolically* in the future. Work on symbolic approaches to model checking can be roughly separated into two strands (see Section 2.4.3): (i) *symbolic model checking* based on the use of *binary decision diagrams (BDDs)* [181], and (ii) *bounded model checking* based on the idea of *satisfiability solving (SAT)*. In the first, BDD-based, approach, large sets of states and transitions are represented symbolically as Boolean formulae and translated into BDDs (compressed version of binary decision trees) which allow for efficient manipulation. BDDs are constructed iteratively which suggests a possible combination with the approach described in this thesis. At the current stage, our focus is purely on the analysis of individual traces; however, similar to the idea of on-the-fly model checking mentioned in Section 10.3.1, the transition relation could be 'learned' as the state space is being explored and a symbolic representation of the state space could be constructed on-the-fly. In the second approach, bounded model checking, a SAT procedure is used instead of BDDs. The idea is to construct a formula which is true if and only if there is a counterexample in the underlying model. To that end, the state space is 'unrolled' for a fixed number of steps (starting with 1), encoded as a Boolean formula, conjoined with the property to be verified and checked for satisfiability. If the resulting formula is not satisfiable, then a counterexample has been found and the property is refuted. Otherwise, the path length is increased. This process is repeated until (i) a witness has been found, or (ii) the maximum path

length has been reached. Similar to BDD-based model checking, the approach is iterative in nature and therefore related with the approach described in this thesis. We aim to investigate the correspondence to both approaches further as part of our future work.

The purpose of the next chapter is to further investigate the idea of sampling through repeated simulation as a means to explore sets of *possible worlds* within the space of the simulation and its usefulness for the verification of advanced types of properties which extend beyond pure safety checking.

Chapter 7

Beyond individual runs: exploring the space

7.1 Introduction

The previous chapter described algorithms for verifying simLTL properties upon fragments of simulation traces as well as a method to estimate the probability of a property given a whole set of traces. In general, when dealing with systems that exhibit a high degree of randomness — as typically is the case for agent-based simulations — individual runs are of little explanatory relevance. In order to cope with random variance, a sufficiently large number of traces needs to be examined in order to get a realistic view on the system's behaviour and increase confidence in the results obtained.

However, apart from just estimating the probability of a single property, the evaluation of properties on multiple trace fragments also offers the opportunity for advanced types of analysis by studying the *relationship* between events. The purpose of this chapter is to explore this idea and describe analyses that can be performed by exploiting the combination of statistical sampling of trace fragments and the verification of simLTL properties upon those traces for the purpose of correctness checking.

As described in Section 2.2.3, correctness against the background of agent-based simulations is hard to define, particularly in the context of validation. This is due to the fact that the type of validation required for a simulation is highly dependent upon its *purpose*. As Schmid puts it, "validation is the process of

determining the sufficient accuracy to which a model or simulation is a representation of the real world system from the perspective of the specific purpose of the model or simulation." [219]. Among the different views expressed in literature [9, 177, 178], we concentrate on the following, in our opinion most general purposes of agent-based modelling: *explanation*, *exploration* and *prediction*. Despite the seemingly clear distinction between the three concepts, there is a surprising level of disagreement or contradiction in the literature. This is mostly due to the fact that these purposes bear no absolute meaning, but instead need to be viewed against the background of the *type* of simulation that they are referring to. In Section 2.2.2, we described a classification according to which agent-based simulation models can be categorised as *generators*, *mediators*, or *predictors* [115]. It is obvious that instances of the three groups differ significantly, both in terms of their inherent characteristics and their associated definition of veracity, validity or accuracy. Clearly, a generator such as Schelling's segregation model has different requirements than a highly data-driven predictive model whose agent population has been initialised with data that has, for example, been obtained through a complex machine learning process on real sets of consumer basket data. It is exactly these differences which exacerbate the interpretation of purposes like explanation, exploration or prediction. The following section discusses the notion of explanation against the background of the different types of models mentioned above.

7.2 The meaning of explanation

Let us start with a closer analysis of the concept of explanation. What does it mean to explain a phenomenon, that is, to find an *explanans* for a given *explanandum*? Clearly, as indicated above, this depends on the type of the model, its purpose and its domain; explanation of a phenomenon in a generator model has a different flavour than in a predictor model.

The nature of explanation also depends on the background of the person attempting it. Statisticians often define explanation as the mere identification of sufficient conditions: explanation reduces to the analysis of correlated variables. As Marks put it, "for prediction, sufficiency suffices. There is no need to know which if any alternate conditions will also lead to the observed endogenous behaviours" [177]. In this context, conditions are defined in terms of correlations between variables. For example, variable A is correlated with B if the probability of A *and* B is significantly higher than their joint probabilities[1]. This clearly makes sense. If the sole purpose of an agent-based simulation is to extrapolate historical trends and if there is enough individual data available that the individual agents can be trained with,

[1] A more comprehensive discussion of sufficiency and necessity is given in Section 7.3.

then there is no purpose in explaining *why* the agents acted as they did. All the modeller needs to do is to make sure that the behaviour of the model represents the historical data with a sufficient degree of accuracy (i.e. shows a good fit). In this case, validation becomes mere line-fitting; correlation becomes the explanans.

It is also clear that the situation is rather different in the case of generator models which can often be found in the social sciences. In general, social scientists pose more strict requirements on the definition of explanation. Instead of mere correlation between variables, they tend to give explanation a *causal* flavour and define it as the identification of *generative mechanisms*, i.e. the 'cogs and wheels' that are able to bring about the phenomenon under consideration [116]. Consider again Schelling's segregation model. Instead of fitting historical data (which would be hard to impossible anyway due to its abstract nature), the model's purpose is to show how a reasonable behavioural hypothesis about individuals can lead to a puzzling and to some extent counterintuitive outcome at the macro level. In this case, explanation means answering the question *why* the overall behaviour emerged which, in turn, requires to unveil the *mechanisms*, the cogs and wheels that lead to the global outcome[2]. In this case, the causal relationship — the mechanism — becomes the explanans.

The issue of the explanatory power of agent-based simulations has been debated extensively in literature [85, 174]. An interesting overview of the epistemological issues surrounding agent-based simulation is given by Epstein in his paper about the foundations of agent-based generative social science [88], as well as by David [67]. According to Epstein, the central contribution of agent-based modelling is its ability to facilitate *generative explanation*. By *growing* a macro-level phenomenon from behavioural assumptions about the individual behaviour, *generative sufficiency* can be attained. However, as Epstein argues, "generative sufficiency is a necessary, but not sufficient condition for explanation". This is due to the fact that, even if an agent-based simulation is capable of bringing about, i.e. generating, a certain macro-level phenomenon, there may be other (even better) models which are equally capable of generating it. In fact, as mentioned above, finding the model which explains the phenomenon as well as possible is the central purpose of validation; it is a continuous quest for a better model[3].

The epistemological issues related with (agent-based) modelling are highly philosophical in nature and need not concern us in further detail here. However, it becomes clear that, in the context of questions of correctness of a simulation, they cannot be fully neglected. If the purpose of the simulation is to explain a phenomenon and the task of validation (internal or external) is to assess whether the simulation satisfies

[2] Consequently, generator models are also often referred to as *mechanistic* models

[3] According to Ripley, validation is not (as often stated) the quest for the *best* model since — especially in the case of generative or mechanistic models — multiple different models may explain a phenomenon equally well [208].

its purpose with a sufficient level of accuracy, then a definition of explanation is necessary in order to define a proper validation technique. We can thus conclude that assessing the validity of an agent-based simulation requires the analysis of its explanatory qualities which, in turn, requires the exposure of internal mechanisms. An important concept in this context is that of an *artefact* [97], i.e. a mechanism which has a causal effect on the observed behaviour where, in fact, it should not. A typical example of an artefact is a particular random number generator significantly influencing the qualitative outcome of the model. The detection of artefacts is a central issue in validation. In order to identify a mechanism as an artefact, one needs to perform two tasks: (i) confirming the existence of the mechanism, and (ii) establishing that it indeed has a causal effect — i.e. *explaining* it. As we shall see below, causal analysis requires the comparison of different models. Changing attributes and parameters and assessing their effect on the model's outcome is also the purpose of exploration. As a consequence, explanation often *requires* exploration.

Let us summarise the ideas discussed so far. In order to assess the correctness of a model, explanation is essential — it is an integral part of the process. However, different types of models yield different definitions of explanation. The type of model determines how deep we need to immerge into the model in order to analyse its dynamics. For a predictive model, the detection of correlated variables and behaviours may suffice. For a generative, explanatory model, the underlying mechanisms may need to be unveiled. In order to accomplish that, causal relationships may need to be detected which, in turn, requires certain exploration of the model space.

The purpose of this the following sections is to describe advanced types of analysis that can be performed on top of the statistical runtime verification approach presented in Chapter 6. More specifically, the following subsections discuss how the interpretation of simLTL formulae on multiple sets of trace fragments or 'possible worlds' can be used to infer higher-level or 'explanatory' information about the dynamics of the simulation in order to support the overall internal and external validation process as illustrated in the previous paragraphs. We start with the interpretation of the probability of a simLTL formula in terms of its truth distribution in Section 7.3 and how this can be used to answer basic properties. The usage of probability analysis for the detection of statistical correlations between complex events is described in Section 7.4. Section 7.5 extends the focus of analysis to the internal structure of complex events and describes how different conditional relationships between complex events — both deterministic and stochastic — can be detected using simLTL model checking. At that point, the ground is laid for a discussion about causality in Section 7.6. The chapter concludes with a summary and a brief discussion of the strengths and weaknesses of the building blocks described.

7.3 Probability analysis

The most straightforward purpose of checking a formula on a set of runs is to estimate its probability of being true. The probability of a simLTL formula ψ — denoted $Pr(\psi)$ — is the measure of all those fragments of a given length that satisfy ψ. For the remainder of this document, will refer to this process as *probability analysis*. An algorithm for estimating the probability of a simLTL formula was described in Section 6.4. Given confidence parameter δ and accuracy parameter ϵ, the estimated probability obtained using function `Estimate` and the actual probability of ψ satisfy the following inequality (irrelevant function arguments are denoted with an asterisk):

$$Pr(\mid Pr^{4}(\psi) - \texttt{Estimate}(*, \psi, \delta, \epsilon) \mid \leq \epsilon) \geq \delta \tag{7.1}$$

In addition to pure replications, fragments of individual simulation runs can also be viewed as *possible worlds* brought about by the behavioural hypotheses built into the underlying model. In this context, each trace fragment represents a state of affairs that *could have been the case*. The notion of possible worlds has been discussed extensively in literature [156] and forms the basis of modal logic [130]. Here, possible worlds form the semantic basis for logical statements involving, for example, *alethic* (necessity, possibility), *deontic* (obligations, permissions) or *epistemic* (knowledge) modalities. Alethic modal logic deals with logical necessity, possibility or impossibility. Modal operators can be understood as quantifying over possible worlds. A logical formula ψ is thus *necessarily* true (denoted with operator '$\Box$') if it is true in all possible worlds. Likewise, ψ is *possibly* true (denoted with operator '$\Diamond$') if there is at least one possible world in which ψ is true. The concepts of possibility and necessity are directly applicable to probability analysis. A formula ψ is necessarily true if it occurs with 100% probability, i.e. if it is true in all trace fragments. Likewise, it is possibly true if it occurs with a non-zero probability, i.e. if there is at least one trace fragment for which it is true:

$$\Box\phi \text{ holds iff } Pr(\psi) = 1.0 \tag{7.2}$$

$$\Diamond\phi \text{ holds iff } Pr(\psi) > 0 \tag{7.3}$$

Note that, due to the approximate nature of the approach described in this work, 100% certainty can never be achieved. To this end, it is more useful to check whether the evaluation results lie within

[4] Remember that, if no subscript value is given, then we assume that the property is being evaluated on the full trace or that the fragment size does not matter for the purpose of description. The latter is the case here.

an acceptable error margin denoted by ϵ. Apart from the modalities of necessity and possibility, the analysis of possible worlds also forms an important basis for the study of counterfactual statements, i.e. statements about what *could have happened*. This issue is discussed in further detail in Section 7.6. Before, however, we turn to the analysis of correlations between complex events in the following section.

7.4 Correlation analysis

Given the capability of checking simLTL formulae on individual runs and an interpretation of a formula's truth across multiple runs as its probability, we can now start to build more complex queries. Probability analysis can be used conveniently as the basis for advanced types of analysis such as *probabilistic dependence* or *correlation*. Correlation is an important concept in statistics. In addition to the detection of correlated variables, the usage of simLTL model checking and probabilistic analysis allows us to go one step further and also detect correlations between events, i.e. between complex and possibly temporally extended phenomena. Positive correlation can therefore be defined as a binary function $posCorr : \Psi \times \Psi \rightarrow \{\text{true}, \text{false}\}$ which accepts two complex events formulated as simLTL formulae $\psi : \Psi$ and returns true if they are positively correlated, otherwise false.

To exemplify positive correlation, let A and B denote two complex events which are represented as simLTL formulae. According to conventional probability theory, the two events are positively correlated if the probability of A and B occurring together (i.e. in the same run) is higher than their joint probability [124], described formally as $Pr(A \wedge B) > Pr(A) \cdot Pr(B)$. Statistical correlation is only meaningful if the analysis is performed on a representative sample of the underlying population. If A and B are agent formulae, it is thus important to check them on randomly selected agents. This is ensured by the semantics of simLTL, according to which unquantified agent formulae are evaluated upon uniformly randomly selected agent traces, as described in Section 5.4.2. The semantics of the $posCorr$ function can then be defined as follows:

$$
\begin{array}{|l}
\hline
posCorr : \Psi \times \Psi \rightarrow \{\text{true}, \text{false}\} \\
\hline
posCorr(A, B) = \text{true} \Leftrightarrow Pr(A \wedge B) > Pr(A) \cdot Pr(B) \\
posCorr(A, B) = \text{false} \Leftrightarrow Pr(A \wedge B) \leq Pr(A) \cdot Pr(B) \\
\hline
\end{array}
$$

The function returns true if and only if events A and B are positively correlated or *dependent* when being evaluated on either simulation traces or uniformly randomly chosen agent traces. In other words, there is a higher chance of an agent exhibiting A and B together rather than separately. The definition of functions for negative correlation and non-correlation, i.e. statistical independence, are omitted; they can be given accordingly.

Given the algorithms described in Chapter 6, positive correlation between two events A and B can be decided as follows (again, irrelevant function arguments are denoted with an asterisk):

$$\texttt{Estimate}(*, A \wedge B, *, *) > (\texttt{Estimate}(*, A, *, *) \cdot \texttt{Estimate}(*, B, *, *)) \tag{7.4}$$

Probabilistic correlation analysis represents an important building block in the quality assurance process. It can give insights into the system's dynamics by revealing behaviours which are coupled, i.e. whose occurrence is (entirely or to some extent) synchronised. Correlation analysis is cheap since the probability of all involved events can be checked on the same set of runs. As we shall see further below, this is not the case for causal analysis. The number of runs necessary in order to prove a causal statement is a function of the number of individual formulae within the property.

It is also important to note that, due to its purely observational nature, correlation analysis does not take into account any temporal ordering of the events involved. Since both events can be temporally extended, they could occur in arbitrary temporal order and with partial or even complete temporal overlap. All that the correlation function expresses is whether there is *observational correlation*, i.e. whether both events happen within the same simulation run.

For predictor models, correlation analysis may, in many cases, be entirely sufficient for the explanation of a particular phenomenon. It is well known, however, that correlation should never be confused with causality. The synchronous occurrence of two events may *indicate* a causal relationship but is never able to prove it (at least not in a strict sense, as briefly described below). For example, due to its negligence of any temporal ordering, events, whilst correlated, may violate the asymmetry principle of causality which states that causes should precede their effects and not vice versa [124]. This issue renders pure correlation analysis unsuitable for the detection of mechanisms and, thus, also the detection of artefacts. However, correlation analysis can be used to detect *symptoms* which can motivate further, more tailored experiments. For example, if A and B are positively correlated, one can be sure that one of the following

three scenarios is definitely true: (i) A is a cause of B, (ii) B is a cause of A or, (iii) there is a common cause for A and B.

Example 7. Consider the following property: "There is a negative correlation between an agent's exposure to media and its infection rate". Assuming the existence of predicates '$exposed$' and '$infected$', this can be formalised as follows:

$$Pr(exposure \wedge \mathbf{F} infected) < Pr(exposure) \cdot Pr(\mathbf{F} infected)$$

The property states that there is a negative correlation between '$exposure$' and '$\mathbf{F}$ $infected$'. Both formulae are agent formulae. Since they are unquantified, their evaluation on a given simulation trace results in an evaluation on a randomly sampled agent trace. This in combination with the evaluation on randomly chosen simulation traces produces a value which can be interpreted as the desired correlation factor.

At this point, it is useful to briefly explore the correspondence between the analysis of temporal properties as described in this work and statistical time series analysis. In statistics, correlation analysis can be generalised to the temporal case by measuring the *internal correlation* or *autocorrelation* of a time series. In this case, the internal association between observations is measured which roughly corresponds to the idea of temporal relationships between events described in this work. However, as opposed to a time series which represents a sequence of simple data points, events described by temporal logic properties can be of arbitrary internal complexity. A statement such as $\phi_1 \Rightarrow \mathbf{F}\phi_2$, for example, describes a simple temporal relationship between events ϕ_1 and ϕ_2. If ϕ_1 and ϕ_2 refer to the value of a simple numeric variables, then certain temporal questions may indeed by answerable by means of time series analysis. However, due to the recursive nature of temporal logic, subformulae may themselves be temporal in nature. Let, for example, $\phi_1 = a \wedge \mathbf{X}b$ and $\phi_2 = \mathbf{G}c$; then we can formulate properties such as $(a \wedge \mathbf{X}b \Rightarrow \mathbf{FG}c)$ which states that, if b holds one time step after a holds, then c will eventually hold forever. Properties of that type cannot be answered through pure time series analysis.

A similar point can be made for statistical tests of causality, e.g. *Granger causality*. Given two time series t_1 and t_2, then t_1 can be said to Granger-cause t_2 if and only if t_1 can be statistically shown to forecast t_2, i.e. if events happening in t_1 also happen temporally delayed in t_2. The correspondence between t_1 and t_2 is typically shown with statistical hypothesis tests. Despite its causal flavour, Granger test can only find "predictive causality" [86]. Furthermore, Granger tests are restricted to the analysis of relationships between time series, i.e. sequences of simple data points. As described above, temporal

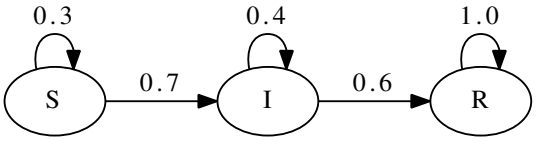


FIGURE 7.1: State transition diagram of a simple SIR model

logic statements allow for the formulation of hierarchical temporal statements and are therefore more expressive. Nevertheless, a combination between (i) observations formulated in temporal logic and measured using verification, and (ii) an analysis of their relationship by means of statistical techniques (e.g. correlations, autocorrelation, or even Granger causality) is clearly beneficial and a central design principle of the approach described in this work. Model checking plus probability analysis provides the foundation for advanced types of analyses, of which correlation analysis is the most basic one. The next section introduces conditional reasoning which allows a modeller to further investigate the internal structure of complex events by detecting *sufficient* and *necessary conditions*.

7.5 Conditional analysis

Probabilistic correlation as described in the previous section observes complex events in isolation and focusses on their joint occurrence. This may be a sufficient explanation for certain types of models (e.g. for purely predictive, highly data-driven models), yet is certainly not for all of them. Especially generator models require further insight into the structure of complex events (i.e. their internal mechanisms) in order to provide deeper and therefore sufficient explanation for certain phenomena and, for example, identify them as artefacts. In this section, we deal with *conditional statements*, i.e. statements that describe the fact that *if* A happens, *then* B (i) should happen, (ii) should not happen, or (iii) should happen only with a certain probability. Due to discrepancies between logical statements and natural language, particular care needs to be taken when conditional statements are translated into logical formulae. This issue is further explicated in Section 7.5.1.2.

Conditional analysis is particularly useful for checking the compliance of an agent (or a group of agents) with its implemented protocol, i.e. its behavioural logic, which involves verifying whether individual state transitions are correct. As a motivating example, consider again the transmission model introduced in Section 5.1.1. Here, an agent is only allowed to transition between three states *Susceptible*

(S), *Infected (I)* and *Recovered (R)* as shown in Figure 7.1. In order to verify the correctness of the implemented behaviour, we thus want to formulate the following properties:

1. *"Whenever an agent is in state S, its successor state will always be either S or I"*
2. *"Whenever an agent is in state I, its successor state will always be either I or R"*
3. *"Whenever an agent is in state R, its successor state will always be R"*

Furthermore, Figure 7.1 shows that transitions are stochastic, i.e. that there is a particular probability of an agent transition into a successor state. In real-world models, these probabilities are often functions of an agent's current beliefs, the state of the environment, its neighbours, etc. For simplicity, however, we assume that probabilities are pre-determined. In order to ascertain whether the agent exhibits the behaviour shown in Figure 7.1, we thus also need to formulate properties about the probability of certain transitions:

1. *"If an agent is in state S, then there is a 30% probability of staying in S and a 70% probability of transitioning into I"*
2. *"If an agent is in state I, then there is a 40% probability of staying in I and a 60% probability of transitioning into R"*
3. *"If an agent is in state R, then there is a 100% probability of staying in R"*

As we shall see below, there is a significant difference between the two types of properties described above — both in terms of their formalisation and in terms of their verification. We refer to the former group as *qualitative conditional* or *compliance properties*, and to the latter group as *quantitative conditional properties*. In order to answer questions of both types, we first need to extend our focus from pure correlation to the analysis of *conditional dependencies* between complex events. We start with qualitative conditional properties.

7.5.1 Qualitative conditional properties

Recall that a qualitative conditional property describes a deterministic relation between events, e.g. values of agent attributes. As described above, a typical application is the assessment of state transitions

within the simulation. Conditional dependencies can be formulated logically using the implication operator '⇒'. Let $A \Rightarrow B$ be a simLTL formula which states that A implies B. By checking the formula on multiple simulation runs, we obtain $Pr(A \Rightarrow B)$. Conditional dependencies between events can be further characterised by means of *sufficiency* and *necessity*. A is a sufficient condition for B if A always (i.e. in all possible worlds) implies B[5]. Conversely, A is a necessary condition for B if B always implies A:

$$\Box(A \Rightarrow B) \text{ holds iff } Pr(A \Rightarrow B) > 1.0 - \epsilon \quad \text{(A is a sufficient condition for B)} \tag{7.5}$$

$$\Box(B \Rightarrow A) \text{ holds iff } Pr(B \Rightarrow A) > 1.0 - \epsilon \quad \text{(A is a necessary condition for B)} \tag{7.6}$$

The analysis of sufficient and necessary conditions can give helpful insight into the dynamics of the system under consideration. However, care should be taken when the results are interpreted: statements about sufficiency and necessity are always relative to the configuration they are being analysed on. It is tempting to generalise the results to the model (or even the real-world domain) but this may be misleading, especially when event A refers to the value of a variable that does not change during the run of a simulation. Imagine, for example, a simple transmission simulation in which all agents are initially (and forever) male. We can formulate the following events:

$$A = \text{being male}$$

$$B = \text{being infected}$$

Further suppose that in every run of the model's current configuration, the agents will always eventually be infected. In this case, both $\Box(A \Rightarrow B)$ and $\Box(B \Rightarrow A)$ are true which proves that being male is both a sufficient and a necessary condition for being infected — a result which is intuitively wrong. A cannot be analysed in terms of sufficiency or necessity because there is no variation in the valuation of the agent's gender attributes and the population is thus not representative. In order to answer questions about sufficiency or necessity of being male, a representatively large population with both male and female agents would be necessary. Logic and verification alone is thus not sufficient for this type of analysis; proper experimental design is equally important.

[5] As mentioned above, in an approximate context, 'always' is difficult to characterise. In order to avoid strictly assuming 100% certainty, we associate 'always' with the probability of an event being greater than 1 minus some (problem-dependent) error margin ϵ.

Furthermore, it is essential to bear in mind that care should be taken when purely logical implications are given a causal meaning. One problem of a pure logical implication such as $A \Rightarrow B$ (i.e. one without temporal operators) is that it does not take into account time, i.e. it does not require the consequent B to happen *after* the antecedent A. In order to give our implications correct meaning, we thus need to make sure that they have correct temporal semantics. In order to solve this problem, we can use the 'leads-to' operator which has been originally proposed by Hansson and Jonsson [114]. It is defined as follows:

$$A \overset{f}{\rightsquigarrow} B \equiv \mathbf{G}(A \Rightarrow \mathbf{F}B) \tag{7.7}$$

The lowercase letter 'f' stands for 'finally' and denotes the future-time flavour of this operator as opposed to its past-time counterpart described below.

Intuitively, the leads-to operator can be understood as extending conventional logical implication ('$\Rightarrow$') with temporal constraints. It describes the fact that, if A happens, B will also *eventually* happen. This implies that B cannot happen before A — the asymmetry of causal relationships (one of their fundamental properties, see further below), is thus preserved. Analogous to the non-temporal case above, sufficiency can be established as follows:

$$\Box(A \overset{f}{\rightsquigarrow} B) \text{ holds iff } Pr(A \overset{f}{\rightsquigarrow} B) > 1 - \epsilon \quad \text{(A is sufficient for bringing about B)} \tag{7.8}$$

In order to check for necessity, we need to 'revert' time, i.e. we need to check whether every time B happens, A happened before. This can be done using the past-time operator '$\mathbf{O}$' ('once') [96]. In order to describe the 'was succeeded by' relationship, we introduce a new operator '$\overset{o}{\rightsquigarrow}$' which is defined as follows:

$$B \overset{o}{\rightsquigarrow} A \Leftrightarrow \mathbf{G}(B \Rightarrow \mathbf{O}A) \tag{7.9}$$

$B \overset{o}{\rightsquigarrow} A$ states that, whenever B happens, A also happened *once*, i.e. at a previous point in time. The same can also be achieved using future-time operators only [176]:

$$B \overset{o}{\rightsquigarrow} A \Leftrightarrow \neg\,(\neg\, A\, \mathbf{U}\, (A \wedge \neg\, B)) \tag{7.10}$$

The fact that A is a necessary temporal condition for B can now be described as follows (the same caution as in the case of necessity in purely logical conditions is advised):

$$\Box(B \overset{o}{\rightsquigarrow} A) \text{ holds iff } Pr(A \overset{o}{\rightsquigarrow} B) > 1.0 - \epsilon \quad \text{(A is necessary for bringing about B)} \tag{7.11}$$

From the definition of sufficient and necessary conditions we can derive the definition of a *temporal biconditional* (operator '$\leftrightsquigarrow$') as follows:

$$A \leftrightsquigarrow B \Leftrightarrow (A \overset{f}{\rightsquigarrow} B) \wedge (B \overset{o}{\rightsquigarrow} A) \tag{7.12}$$

Informally, a temporal biconditional states that A is always temporally followed by B and that B is always temporally preceded by A. It can thus be seen as the temporal equivalent to the logical biconditional described above. It is also related with correlation.

Instead of stipulating that, if A happens, B will *eventually* also happen, it is often necessary to describe events which happen *strictly consecutively*. In this case, 'finally' can be replaced with 'next' (denoted 'x') and 'once' can be replaced with 'previously' (denoted 'p') as follows.

$$A \overset{x}{\rightsquigarrow} B \Leftrightarrow \mathbf{G}(A \Rightarrow \mathbf{X}_{\forall} B) \tag{7.13}$$

$$B \overset{p}{\rightsquigarrow} A \Leftrightarrow \mathbf{G}(\mathbf{X}_{\forall} B \Rightarrow A) \tag{7.14}$$

Note that, due to the wrapping of a statement involving the 'next' operator into the 'globally' operator, the weak version of 'next' needs to be used in order to avoid the evaluation of the formula to false in the final state of the trace. Alternatively, the same questions can be answered by evaluating the following simplified formulae on trace fragments of size 2[6]:

$$A \overset{x}{\rightsquigarrow} B \Leftrightarrow A \Rightarrow \mathbf{X}B \tag{7.15}$$

$$B \overset{p}{\rightsquigarrow} A \Leftrightarrow \mathbf{X}B \Rightarrow A \tag{7.16}$$

[6]Remember that we denote with an unquantified '$\mathbf{X}$' the fact that both the weak and the strong version can be used.

7.5.1.1 Application: Verifying state transitions

As indicated above, conditional analysis is particularly helpful for verifying the correctness of state transitions within the simulation, both on the agent and on the group level. As mentioned above, qualitative conditional properties are also referred to as *compliance properties*. We thus refer to this type of analysis as *compliance checking*.

Example 8. Consider the transmission simulation introduced above in which agents are only allowed to transition between three values for attribute 'health', *Susceptible (S)*, *Infected (I)*, and *Recovered (R)*, as shown in Figure 7.1. The informal requirement that no transition between S and R, I and S, R and S or R and I should ever occur can now be formalised as a set of conditional properties as follows (note that the properties need to be evaluated on trace fragments of length 2):

$$\Box\left((health = S) \overset{x}{\rightsquigarrow} \neg\left(health = R\right)\right) \tag{7.17}$$

$$\Box\left((health = I) \overset{x}{\rightsquigarrow} \neg\left(health = S\right)\right) \tag{7.18}$$

$$\Box\left((health = R) \overset{x}{\rightsquigarrow} \neg\left(health = S \vee health = I\right)\right) \tag{7.19}$$

The properties are formulated by specifying which events *should not* happen given the antecedent, i.e. the conditioning event has been observed. For example, the first property in the list above expresses that, whenever an agent is susceptible, it must never transition into the $Recovered$ state within one step. Alternatively, the properties can be formulated in an 'inverted' way, i.e. by specifying which events *should* happen given the conditioning event has been observed:

$$\Box\left((health = S) \overset{x}{\rightsquigarrow} \left(health = S \vee health = I\right)\right) \tag{7.20}$$

$$\Box\left((health = I) \overset{x}{\rightsquigarrow} \left(health = I \vee health = R\right)\right) \tag{7.21}$$

$$\Box\left((health = R) \overset{x}{\rightsquigarrow} \left(health = R\right)\right) \tag{7.22}$$

These properties express that a local state must *always* be succeeded by a successor state as described by the protocol in Figure 7.1. Property 1, for example, states that, whenever an agent is susceptible, it must either remain so or transition into the 'Infected' state.

Another approach is to formulate the statement above as a conditional property which describes a situation that should *never* happen. One might be tempted to translate the statements above (which are, essentially, 'globally' formulae) into their negated counterparts as follows:

$$\neg \Diamond \mathbf{F}((health = S) \Rightarrow \mathbf{X}_{\forall}(health = R)) \tag{7.23}$$

$$\neg \Diamond \mathbf{F}((health = I) \Rightarrow \mathbf{X}_{\forall}(health = S)) \tag{7.24}$$

$$\neg \Diamond \mathbf{F}((health = R) \Rightarrow \mathbf{X}_{\forall}(health = S \vee health = I)) \tag{7.25}$$

These properties state that it is *never possible* to reach a state from which a non-allowed transition according to the protocol shown in Figure 7.1 is performed. At first sight, this way of describing the requirements seems correct. As discussed in the following subsection, however, the direct association of our natural language requirements with properties 7.23 – 7.25 is problematic as a consequence of which the verification would produce misleading results.

7.5.1.2 Interpreting conditional statements

A general problem with logical implication is the possibility to prove intuitively counterintuitive statements. These issues are known as the *paradoxes of material implication* [81]. One particular famous example, the *paradox of entailment*, which results from the principle of explosion in classical logic, states the puzzling fact that, whenever the antecedent of a logical implication is false, anything can be deduced ("*ex falso quodlibet*"). For example, if both A and B are false, then $A \Rightarrow B$ is true. This is clearly problematic as it renders statements such as *"if the moon is made of green cheese then life exists on other planets"* true.

This also represents a problem for verification and validation. The paradoxes of material implication make it possible to derive insights into the behaviour of the simulation which are intuitively incorrect. For example, if we assume that A necessarily leads to B, i.e. $\Box(A \overset{f}{\rightsquigarrow} B)$, and we check this property on n simulation runs in each of which neither A nor $\mathbf{F}\ B$ happens, then the evaluation would still return $true$ and thus prove that A *does* necessarily lead to B. This, however, is intuitively wrong since neither A nor B ever happened in the observed simulation runs. Consequently, no statement at all should be made in this case. But this is also problematic since it violates the principle of bivalence inherent to two-valued logic.

This is also critical for the verification of negated compliance properties such as those in 7.23 – 7.25. Subformula '$(health = S) \Rightarrow \mathbf{X}_{\forall}(health = R)$' is true whenever the antecedent and the consequent is true *or* whenever the antecedent is false (that is, has not occurred). In a model where the antecedent never occurs, '$\mathbf{F}((health = S) \Rightarrow \mathbf{X}_{\forall}(health = R))$' will thus always be true which, in turn, renders the entire negated property false and thus falsely indicates a violation of the formulated requirement.

The core problem with the paradoxes of material implication is the fact that truth in a logical sense does not necessarily correspond with the notion of truth in natural language. The paradox of entailment implies that, for example, from a contradiction (which is always false) anything can be inferred. It is obvious that, in natural language, this type of inference can be problematic. Logical implication is often confused with different meanings of 'if ... then' in natural language. One way to circumvent these problems is to use non-classical logics such as relevant logic [175] which, however, shall not be further discussed here. Another solution is to give up logical bivalence and follow a two-step process: First, the existence of the antecedent needs to be checked. If this is successful (i.e. the antecedent has actually happened and can thus be observed), then conditional analysis can be performed. If the antecedent cannot be observed, then no statement about the conditional relationship between the antecedent and the conclusion can be made. In order to be able to refuse a statement, we would need to allow for *'don't know'* or *'undefined'* as an additional return value alongside $true$ and $false$ when checking a simLTL formula in order to denote that the property cannot be answered [92]. Again, however, this would violate the principle of bivalence in simLTL. A third solution is to translate the conditional statement into a statement in which implication is replaced by conjunction. Consider again Example 8. The conditional properties can be translated into unconditional ones as follows:

$$\neg \Diamond \mathbf{F}((health = S) \wedge \mathbf{X}_{\forall}(health = R)) \tag{7.26}$$

$$\neg \Diamond \mathbf{F}((health = I) \wedge \mathbf{X}_{\forall}(health = S)) \tag{7.27}$$

$$\neg \Diamond \mathbf{F}((health = R) \wedge \mathbf{X}_{\forall}(health = S \vee health = I)) \tag{7.28}$$

Despite their apparent similarity, Properties 7.26 - 7.28 differ significantly from Properties 7.23 - 7.25. Due to the former's use of logical AND instead of implication, the problem of falsely reporting verification failure is avoided. As before, it is possible to answer the same question on trace fragments of length 2 which allows for the following simplification:

$$\neg \Diamond((health = S) \wedge \mathbf{X}(health = R)) \tag{7.29}$$

$$\neg \Diamond((health = I) \wedge \mathbf{X}(health = S)) \tag{7.30}$$

$$\neg \Diamond((health = R) \wedge \mathbf{X}(health = S \vee health = I)) \tag{7.31}$$

7.5.1.3 Necessity and sufficiency

The idea of necessity and sufficiency can also be helpful for the verification of state transitions. The compliance properties given above ascertain that, starting in a particular state, only transitions into correct successor states happen. This relates to a sufficiency check. If $A \Rightarrow B$ holds then A is sufficient for bringing about B. Likewise, if '$(health = S) \Rightarrow \mathbf{X}_{\forall}(health = R)$' holds, then being in state *Infected* is sufficient for transitioning into state *Recovered*. In order to check the correctness of state transitions exhaustively, however, we also need to ascertain the opposite, i.e. that states are only transitioned into from correct predecessor states. This represents a necessity check. The respective properties can be formulated as follows:

$$\Box\,((health = S) \overset{p}{\rightsquigarrow} (health = S)) \tag{7.32}$$

$$\Box\,((health = I) \overset{p}{\rightsquigarrow} (health = I \vee health = S)) \tag{7.33}$$

$$\Box\,((health = R) \overset{p}{\rightsquigarrow} (health = R \vee health = I)) \tag{7.34}$$

For example, Formula 7.32 states that, whenever the successor state is *Susceptible*, then the current state is also *Susceptible*. According to that property, state *Susceptible* can only be transitioned into from itself which, given the protocol shown in Figure 7.1, is clearly correct.

We now have mechanisms to obtain the probability of an arbitrary simLTL property and to describe and detect conditional dependencies between events. For the latter, we can also distinguish between sufficient and necessary conditions and biconditionals. This in combination with the expressiveness of simLTL allows for the formulation of complex correctness properties. In summary, qualitative conditional statements can be interpreted in two different ways: first, they can be understood as properties which express *desired* conditional relationships and whose verification returns true if the requirement has *either been satisfied or not been violated*. However, as indicated above, care needs to be taken when interpreting the results. Due to their reliance upon logical implication, conditional properties are generally susceptible to misleading interpretation due to the principle of explosion which clashes with our intuitive understanding of 'if ... then' statements in natural language. Second, conditional statements can also be formulated as properties which express those conditional relationships that must *never* be

present in the simulation. Special care needs to be taken in this case since the negation of logical conditional statements may produce counterintuitive results. In case of negation, implication statements should thus be translated into conjunctions.

7.5.2 Quantitative conditional properties

Qualitative conditional properties allow for the formulation of conditional relationships between events. It has been shown above how this can help to prove compliance with the implemented finite state model. In many cases, however, transitions are stochastic (as for example in Figure 7.1) and it is thus a natural requirement to also integrate the probability of transitioning between two states into the verification and validation process. As described in Section 2.2.3, the verification of state transitions represents an important tool for the detection of potential causal relationships — the main purpose of internal validation. This is illustrated practically in the case study described in Chapter 9.

The difficulty with the verification of state transitions lies in their fine-grained nature: given that the same state transition may occur (i) not at all, (ii) once, or even (iii) multiple times on a single trace, it is not possible to answer a state transition question by formulating a simLTL property which is evaluated on a single trace in a purely binary fashion, i.e. such that it yields a clear yes/no answer. Consider, for example, the following four (highly stylised) traces which, we assume, have been produced by a simulation:

$$S \rightarrow S \rightarrow I \rightarrow I \rightarrow I \rightarrow R \rightarrow R \rightarrow R \rightarrow ... \tag{7.35}$$

$$S \rightarrow S \rightarrow S \rightarrow S \rightarrow I \rightarrow I \rightarrow I \rightarrow R \rightarrow ... \tag{7.36}$$

$$S \rightarrow I \rightarrow I \rightarrow I \rightarrow I \rightarrow R \rightarrow R \rightarrow R \rightarrow ... \tag{7.37}$$

$$S \rightarrow I \rightarrow I \rightarrow I \rightarrow I \rightarrow I \rightarrow I \rightarrow I \rightarrow ... \tag{7.38}$$

...

According to the protocol shown in Figure 7.1, the transition between $Susceptible$ and $Infected$ has probability 0.7. The goal is to use simLTL together with the evaluation algorithms described in Chapter 6 in order to verify that the output of the simulation complies with the protocol. Let us first look at out-transitions of state $Susceptible$. In Trace 7.35, two out-transitions occur: one into $Susceptible$ and one into $Infected$. In Sequence 7.36, four out-transitions occur: three into $Susceptible$ and one into

Infected. In both Trace 7.37 and Trace 7.38, one out-transition into *Infected* occurs. Since there are traces which both satisfy and do not satisfy the transition of interest (and that even to a varying degree), it is obvious that a binary check on a single trace will not be of much help. Instead, we need, for each trace, to determine the *ratio* of those transitions from *Susceptible* to *Infected* to those from *Susceptible* to *any* successor state. Evaluated and averaged over multiple runs, the resulting value should then converge to the original probability of 0.7. In the example above, Trace 7.35 has ratio 0.5, Trace 7.36 has ratio 0.25 and Traces 7.37 and 7.38 both have ratio 1.0. On average, the ratio is thus 0.6875 which (ignoring the small sample size) suggests that the transition probability of the simulation complies with the one in the specification.

Before thinking about how this value can be estimated using the algorithms described in Chapter 6, we first need to formulate appropriate simLTL properties. The ratio that we are interested in can be intuitively described as the number of times the agent is in state *Susceptible* and transitioning into *Infected* divided by the number of times the agent is in state *Susceptible* and transitioning anywhere. We thus need two properties: one describing the transition from *Susceptible* to *Infected* and one simply describing the agent's residence in state *Susceptible*:

$$\psi_1 = (health = S) \wedge \mathbf{X}(health = I) \tag{7.39}$$

$$\psi_2 = (health = S) \tag{7.40}$$

Let $\psi =$"The transition probability from *Susceptible* to *Infected* is 25%" denote our ultimate property of interest. It can be obtained from the individual probabilities of ψ_1 and ψ_2 as follows:

$$Pr(\psi) = Pr_2(\psi_1)/Pr_2(\psi_2) \tag{7.41}$$

In order to determine the correct probability for ψ, we first need to determine the individual probabilities for ψ_1 and ψ_2 on each trace and calculate their average. We start with ϕ_1 and exemplify the calculation with a simplified version of Trace 7.36 above, namely $\langle S, S, S, S, I, I, I, R\rangle$. Remember that, internally, function `Estimate` calls function `CheckTrace` once for each trace fragment of length n. Since we have to deal with a 'next' formula, the required length of trace fragments is 2. The following set represents all possible trace fragments of length 2:

$$\{\langle S, S\rangle, \langle S, S\rangle, \langle S, S\rangle, \langle S, I\rangle, \langle I, I\rangle, \langle I, I\rangle, \langle I, R\rangle\} \tag{7.42}$$

The evaluation of `CheckTrace` on each of the fragments returns the following values:

$$\texttt{CheckTrace}(\langle S, S\rangle, \psi_1) = \text{false} \tag{7.43}$$
$$\texttt{CheckTrace}(\langle S, S\rangle, \psi_1) = \text{false} \tag{7.44}$$
$$\texttt{CheckTrace}(\langle S, S\rangle, \psi_1) = \text{false} \tag{7.45}$$
$$\texttt{CheckTrace}(\langle S, S\rangle, \psi_1) = \text{true} \tag{7.46}$$
$$\texttt{CheckTrace}(\langle I, I\rangle, \psi_1) = \text{false} \tag{7.47}$$
$$\texttt{CheckTrace}(\langle I, I\rangle, \psi_1) = \text{false} \tag{7.48}$$
$$\texttt{CheckTrace}(\langle I, R\rangle, \psi_1) = \text{false} \tag{7.49}$$
$$\Rightarrow \texttt{SatR}(\langle \text{false}, \text{false}, \text{false}, \text{true}, \text{false}, \text{false}, \text{false}\rangle) = \frac{1}{7} \approx 0.1429 \tag{7.50}$$

Line 7.50 shows the result of the overall call to `SatR` which represents the success ratio of formula ϕ_1. The evaluation of formula ϕ_2 is shown below:

$$\texttt{CheckTrace}(\langle S, S\rangle, \psi_1) = \text{true} \tag{7.51}$$
$$\texttt{CheckTrace}(\langle S, S\rangle, \psi_1) = \text{true} \tag{7.52}$$
$$\texttt{CheckTrace}(\langle S, S\rangle, \psi_1) = \text{true} \tag{7.53}$$
$$\texttt{CheckTrace}(\langle S, S\rangle, \psi_1) = \text{true} \tag{7.54}$$
$$\texttt{CheckTrace}(\langle I, I\rangle, \psi_1) = \text{false} \tag{7.55}$$
$$\texttt{CheckTrace}(\langle I, I\rangle, \psi_1) = \text{false} \tag{7.56}$$
$$\texttt{CheckTrace}(\langle I, R\rangle, \psi_1) = \text{false} \tag{7.57}$$
$$\Rightarrow \texttt{SatR}(\langle \text{true}, \text{true}, \text{true}, \text{true}, \text{false}, \text{false}, \text{false}\rangle = \frac{4}{7} = 0.5714 \tag{7.58}$$

If we divide the two success ratios, we obtain the success ratio of ψ on trace $\langle S, S, S, S, I, I, I, R\rangle$ which, in this case, is $0.1429 \text{ div } 0.5714 \approx 0.25$. In order to estimate the ultimate probability of ψ, we need to perform the same calculation on all traces (7.35 et seqq.) and calculate the average of the resulting values.

So far, the focus of analysis has been on a single configuration of the system, i.e. on a single set of runs produced by a particular configuration of the simulation. All three types of analysis discussed so far — probability, correlation and conditional analysis — are *passive* in the sense that they are based on pure observation of the simulation model: the parameters are fixed and the simulation is executed n times in order to produce sample runs whose frequencies reflect the probability distribution built into the model. By selecting a sufficiently high value for n, we can obtain good estimates of the real probabilities and thus get a good view of the internal dynamics. Correlations between events and conditional dependencies — sufficient an explanation as they may be themselves — can also give indications for true causal relationships. As described in the next section, however, a single configuration can never be sufficient for *proving* causal relationships. In order to do that, we need to go beyond the state space of a single configuration and start to explore different configurations or, in some cases, even different models.

7.6 Causal analysis

Central to explanatory models is their capability to not just show *that*, but also *why*, particular things happen. As mentioned above, correlation-based analysis can be used to *indicate* causal relationships but, due to its passive nature, it is never able to strictly prove their presence. This section focusses on causal analysis as a further type of analysis which can be performed using the framework described in this work. We start with a brief (and, due to the significance of the topic, only very superficial) overview of causality theories in Section 7.6.1, followed by a description of the idea of causal analysis by means of *active intervention* in Section 7.6.2. The section concludes with a description of how the verification framework presented in this work can be used to perform certain types of causal analysis in Section 7.6.3.

7.6.1 Theories of causality

The problem of causality has been discussed extensively throughout history. David Hume's regularity theory marks a famous starting point in the formal treatment of causation [129]. According to this theory, *"a cause is an object, followed by another, such that all objects similar to the first are followed by objects similar to the second. Or in other words, where the first object had not been, the second never had existed"* [66]. Thus, two events A and B can be considered causally related if they always occur together and the assumed cause is always succeeded by the assumed effect.

Over the centuries, a number of criticisms against pure regularity theories have been raised. The following list summarises some of the well-known difficulties which gave rise to the development of alternative theories of causation [124]:

Imperfect regularities: In many cases, causes invariably having to be followed by their effects is too strict a requirement. For example, if we only accepted this strict notion of regularity, then smoking would not be a cause of lung cancer.

Irrelevance: Not every condition A which is always followed by another condition B is necessarily relevant for B. For example, salt that has been hexed by a sorcerer invariably dissolves when placed in water. Nevertheless, hexing is clearly irrelevant for the dissolution of salt.

Asymmetry: Causes always precede their effects but effects never precede their causes. Some theories based on pure regularity (e.g. strict correlation) may violate the asymmetry condition as we shall see below[7].

Spurious regularities: Event A invariably followed by B may *indicate* a causal relationship which, however, need not necessarily exist. For example, the crow of the rooster is regularly followed by the sunrise but it is clear that the rooster's crow does not *cause* the sun to rise. The source of spurious regularities are confounding events, i.e. common causes which are responsible for both A and B to happen together.

As a response to these problems — particularly the problems of imperfect and spurious regularities — different alternative theories of causality have been developed. Instead of strictly requiring the common occurrence of causes and effects, the common assumption underlying *probabilistic theories of causation* is that *causes raise the probabilities of their effect*. A naïve attempt to formalise this idea is to use conditional probabilities as follows:

$$Pr(B \mid A) > Pr(B) \tag{7.59}$$

This inequality states that the probability of B given that A happens is higher than just the probability of B. As opposed to pure correlation, conditional probabilities allow us to make stronger statements about the *effect of one event on another*. Comparing conditional probabilities, for example, allows for

[7] There are theories which state that causes are simultaneous and reciprocal with their causes [217]. This discussion is beyond the scope of this work.

the formulation of statements like the following: *"B is more likely to happen if A happens"* which ultimately leads to the analysis of causes and effects. It is obvious, however, that this solution violates the principle of asymmetry since it allows cause and effect to occur in any order. We can fix this problem by making the following change:

$$Pr(A \overset{f}{\rightsquigarrow} B) > Pr(B) \tag{7.60}$$

Here, the leads-to relationship ensures that B cannot happen before A. Unfortunately, this solution still poses a problem. As mentioned above, an implication is true whenever its antecedent is false. As a consequence, if A never occurs in a set of simulation runs then $A \overset{f}{\rightsquigarrow} B$ always holds and inequality 7.60 will thus be satisfied. Kleinberg and Mishra, whose work takes its inspiration from the work of Suppes [229] and Eells [82], give a probabilistic account of causality using temporal logics which circumvents this problem [140]. According to the authors, A is a *prima facie cause* of B if the following two conditions are satisfied[8]:

$$Pr(\mathbf{F}\ A) > 0 \tag{7.61}$$

$$Pr(A \rightsquigarrow B) > Pr(\mathbf{F}\ B) \tag{7.62}$$

A is a prima facie cause of B if (i) A is reachable with a non-zero probability, and (ii) the probability of A being temporally followed by B is higher than the probability of reaching B. This captures the idea of A raising the probability of B. It also solves the problem of an implication with false antecedent by explicitly requiring that the probability of eventually reaching A is non-zero.

This definition satisfies the axioms of irregularity and asymmetry, yet it still leaves open the possibility for irrelevance and spurious regularities. To this end, Kleinberg and Mishra propose a hypothesis testing approach which helps to determine how significant each cause c is for its effect e by taking into account the *average difference in probabilities* $\epsilon_{avg}(c, e)$ that each c has on e. A problem with this approach is that the set of possible causes X needs to be known in order to calculate $\epsilon_{avg}(c, e)$. Furthermore, given the possibly vast size of X, the hypothesis testing process may become a serious bottleneck.

[8] We give a slightly simplified description. In the authors' original paper, temporal bounds which ensure that the effect must not happen before a minimum and after a maximum number of time units after the cause are also included.

Kleinberg and Mishra's elimination of spurious relationships using hypothesis testing is necessary since they aim to discover causal relationships in existing, purely observational data whose structure is initially entirely unknown. As described above, the leads-to relationship helps to prevent potential causes and effects from being symmetric which strengthens the actual causal significance of prima facie causes. Nevertheless, the inference of causal relationships from purely observational data remains critical. From observational data, we can draw inferences about probabilities and conditional relationships between probabilities. These inferences may give insight into the causal structure of the underlying system. Notwithstanding, unless we *intervene*, the actual causal structure that gives rise to certain effects remain untouched and possibly uncovered. This means that, in order determine whether A causes B, it is not sufficient to merely *observe* the system and check whether the occurrence of A increases the probability of B. Instead, we need to check whether *doing* A increases the probability of B. These ideas can be attributed to Pearl who argued that causality analysis is an inherently manipulatory task [198]. He likened conditional probabilities such as $Pr(B \mid A)$, i.e. the probability of B given that A, with $Pr(B \mid \textbf{see}(A))$, i.e. the probability of B given that A has been *observed*. In order to check whether A has an influence on B, however, we need $Pr(B \mid \textbf{do}(A))$, i.e. the probability of B given that A has been *done*. Changing the course of action by manipulating values intentionally (e.g. explicitly forcing A to happen in our example), is called an *intervention*.

Interventions are at the heart of randomised experiments. By varying conditions between experimental and control groups, causal relationships can be identified. It is obvious, however, that in purely observational studies, interventions are not possible. Since interventions cannot be performed retrospectively, causal relationships need to be derived from existing data sets in this case. The situation is even more challenging in the presence of *counterfactual* statements, which Shpitser and Pearl sharply distinguish from causal statements [223]. Whereas causal statements involve *active intervention* (*'what happens if'*), counterfactual statements involve assumptions about *hypothetical intervention* (*'what would have happened if'*). The authors argue that counterfactual statements are hard or even impossible to prove or disprove in physical experiments; after all, *"we simply cannot perform an experiment where the same person is both given and not given treatment"* [223].

Pearl found a solution to these problems. In his influential book, he describes a technique by means of which the effects of interventions can be studied by combining observational data and a *structural causal model*, a formal model of qualitative assumptions about causes and effects [198]. He was awarded the Turing Prize for his work on reasoning under uncertainty.

Before discussing their relevance for the verification of agent-based simulations, let us summarise the ideas from the previous paragraphs. In order to check for causal relationships, a number of directions can be followed. The purely passive, observational approach based on Suppes' and Eells' work and combined with temporal logic by Kleinberg and Mishra requires the comparison of conditional probabilities with unconditional ones. In order to account for spurious causes, the significance of a potential cause for its effect needs to be assessed which strengthens its causal relevance. Nevertheless, the analysis remains purely observational which is problematic since, as Pearl argues, true causal inference requires active intervention. He emphasises the conceptual difference between *passively observing* that A happened and *actively forcing* A to happen before observing the effect on B and stresses that the latter is the correct approach. This, however, might be difficult to impossible in real-world experiments, either because of moral objections (e.g. forcing people to smoke) or because of the impossibility to retrospectively change the course of history.

7.6.2 Causal analysis with interventions

Fortunately, despite the similarity between agent-based simulations and real-world experiments, there are also significantly differences. At this point, it is important to briefly revisit the distinction between causal and counterfactual statements made by Shpitser and Pearl [223] and briefly mentioned above. According to them, a counterfactual statement is a statement about a hypothetical situation which *could have* but *has not* occurred. In reality, counterfactual statements are ubiquitous. What would have happened if the Second World War had not happened? What would have happened if the weather had been better today? What would have happened if company X had spent more money on advertisement? These questions are all hypothetical in nature. Answering them is hard since we cannot travel back in time and change the course of history.

The situation is different in a simulation context. Here, it *is* possible to travel back in time, change certain conditions and see 'what would have happened if'. If we want to analyse the influence that an event A has on the occurrence of event B, we simply need to control for A and assess the effect on B. As a consequence, every counterfactual statement can be turned into a causal one. Every hypothetical situation can be constructed. We thus employ the following definition of causal influence:

Definition 1. Event A has a causal influence on event B if and only if intentionally forcing A to happen significantly increases the probability of B.

This implies that, in order to assess the causal effect that A has on B, we need to analyse the outcome of the simulation both in the presence of A as well as in its absence and compare the results. If the difference in probability of event B is significant, then we can conclude that A has in fact a causal effect on B. 'Changing something' has an important implication on the notion of the underlying model. If the variable to be changed is a parameter (i.e. a value that is supposed to be changed), then we obtain a new *configuration*. If the variable is an attribute (i.e. a value that is *not* supposed to be changed), however, we obtain a new *model*. This corresponds with the idea of validation as 'the continuous quest for a better model' (see Section 7.2). As a consequence, performing interventions invariably requires a modeller to leave the space of a single configuration and, depending on the actual intervention, either move to a different configuration or even to a different model.

In order to incorporate causal analysis into our framework, we first distinguish between what Pearl refers to as *the probability of sufficiency* (PS) and *the probability of necessity* (PN) [197]. Translated into the world of agent-based simulation, they can be described as follows. Imagine a situation in which a particular configuration c exhibits both the potential cause A and the effect B. We can now ask the following questions: What would have happened if A had not been the case? Would B have occurred anyway? In this case, we assess the presence of a potential cause and an effect in order to ask a counterfactual question about the absence of the cause — we aim to determine whether A is *necessary* for B. This is referred to as PN. Now imagine a situation in which neither A nor B happens in the simulation run. We can now ask the following questions: What would have happened if A had been the case? Would B have happened then? In this case, we aim to determine whether A is *sufficient* for B. This is referred to as PS.

The notion of PS and PN poses an important requirement on our approach: we need to be able to both *activate* an event A (in case of its absence) and *deactivate* an event A (in case of its presence) in order to assess its causal effect on another event B. This can be done through interventions as suggested by Pearl and elucidated above. However, the requirement for actively forcing a potential cause to happen imposes some restrictions on its characteristics: we cannot, in general, actively force emergent or temporally extended events to happen in an agent-based simulation. For example, if we want to investigate the effect that at least 50% of the agents finally knowing about a product (event A) has on the overall sales (event B), we clearly cannot actively force A to happen since it is not explicitly built into the logic of the model but instead emerges from the individual rules.

Manipulations can be of various kinds and may, for example, comprise simple changes in parameter or attribute values, in the addition of actions as part of an agent's or the environment's behavioural logic,

the exchange of environmental components such as the pseudo random number generator being used or complex structural changes such as enabling or disabling the environmental influence or changing the topology based on which agents are connected to each other. Due to this variety and the resulting difficulty in unification, defining a strict formal notation for interventions is of little use. We will instead follow a different path and describe manipulations informally, associate them with *configurations* and give them a unique identifier with which they can be referred to in the final property.

This is best illustrated with an example. Imagine a scenario in which one wants to prove that the initial infection state of an individual agent a_1 (stored in attribute $health$) has a causal influence on the emergence of an observed phenomenon A, e.g. a significant fraction of the population eventually becoming infected. We can now define the following two interventions (the unique identifiers are on the left and the semi-formal description of the actual manipulation is on the right hand side):

$$Iv_1 \hat{=} (a_1.health := infected) \tag{7.63}$$

$$Iv_2 \hat{=} (a_1.health := susceptible) \tag{7.64}$$

The first manipulation sets the initial infection state of agent 1 to *infected*, the second to *susceptible*. Based on Pearl's idea of active intervention and by referring to the unique identifiers of the interventions, the hypothesis about the causal influence can then be formulated as the following property:

$$|Pr(\mathbf{F}\ A \mid Iv_1) - Pr(\mathbf{F}\ A \mid Iv_2)| \leq thr \tag{7.65}$$

This reads as follows: the probability of finally reaching A given that intervention Iv_1 has been done (i.e. agent 1 is infected) differs from the probability of finally reaching A given that intervention Iv_2 has been done (i.e. agent 1 is not infected) by at most t. We have deliberately chosen to allow for a small difference thr in the resulting probability in order to take into account numerical inconsistencies.

With this formalisation at hand, we can define causal influence as a function which accepts a simLTL formula $\psi \in \Psi$, two interventions $Iv_1, Iv_2 \in Iv$ and a real-valued number thr denoting the error threshold:

$$\begin{array}{|l} causalInf : \Psi \times Iv \times Iv \times \mathbb{R} \rightarrow \{\text{true}, \text{false}\} \\ \hline causalInf(\psi, Iv_1, Iv_2, thr) = \text{true} \Leftrightarrow |Pr(\psi \mid Iv_1) - Pr(\psi \mid Iv_2)| \leq thr \\ causalInf(\psi, Iv_1, Iv_2, thr) = \text{false} \Leftrightarrow |Pr(\psi \mid Iv_1) - Pr(\psi \mid Iv_2)| > thr \end{array}$$

where Iv denotes the (infinite) set of all possible interventions. The function returns $true$ if the difference in probabilities between the checks of ψ in the two models is less than or equal to t, otherwise false.

7.6.3 Runtime verification and causal analysis

We can use the runtime verification algorithms described in Chapter 6 to perform the type of causal analysis described above. To this end, consider again the intervention example with the infection state. We assume that Iv_1 corresponds with a configuration of the underlying simulation which implements the intervention logic and is associated with id 1, and Iv_2 corresponds with a configuration associated with id 2. Given error threshold thr, the infection state of agent 1 has no influence on the truth of ψ if the following holds (again, irrelevant function arguments are denoted with an asterisk):

$$\mid \texttt{Estimate}(1, \psi, *, *) - \texttt{Estimate}(2, \psi, *, *) \mid \leq thr \tag{7.66}$$

The association of interventions with configurations that can be referred to by means of a simple id helps to sustain a *separation of concerns* by hiding the intervention logic in the underlying procedural part of the model description and restricting property definition to the declarative aspects.

Through interventions, certain causal relationships can be investigated by controlling for potential causes and observing the effects. By both applying and not applying an intervention to *exactly the same* base model (i.e. without changing anything else, including the seed for the pseudo random number generator), truly counterfactual analyses can be conducted. It is important to note, however, that the causal insights into the simulation are constrained to the model space and may not be generalisable to reality in a straightforward way. Models are always simplifications, abstractions of reality and as such inherently inaccurate. Causal relationships present in the model thus need not necessarily be present in the real world. It is also important to note that proper causal analysis represents experimentation and thus needs to be accompanied with solid experimental design which involves the definition of a hypothesis and a structured approach to the definition of interventions. The nature of the experimental design is highly problem-dependent and beyond the scope of this work; relevant information can, for example, be found in the literature on simulation validation [177].

It is important to mention that causality is a comprehensive topic which has been debated extensively throughout the last centuries. We have thus only touched upon it very superficially and ignored important issues such as, for example, *preemption* [182]. The approach to causal analysis described above is by no means exhaustive; rather than allowing for the automated answering of complex causal claims, we aim to provide a set of basic building blocks using which typical correctness questions — also those with a causal flavour — can be assembled and answered in a runtime verification context.

7.7 Summary

This chapter described the evaluation of simLTL formulae on *multiple simulation runs* and the *implications on the interpretation* of the results. Evaluating a simLTL formula on multiple runs allows us to exceed the scope of verification far beyond pure safety analysis. By exploring the state space, *different types of analyses* can be conducted: *probability*, *correlation*, *conditional* and *causal analysis*. The former three types of analyses are based on pure observation; they may be performed for their own sake or to infer insights into potential causal relationships between events whose existence can then be ascertained using causal analysis. Causal analysis itself requires active *intervention* by manipulation of attributes or actions, either individually or collectively. In order to keep property formulation descriptive, we associated the notion of manipulations with *configurations* which encapsulate the actual changes to attributes or parameters in the procedural part of the model description.

It is important to note that each type of analysis has its strengths and weaknesses: pure probability analysis is inherently limited to the analysis of a single event; correlation analysis ignores temporal or logical relationships between the events under consideration and is restricted to those events happening within the current configuration — a restriction which also applies to conditional analysis. On the other hand, the strength of purely observational analyses lies in their possibility to analyse relationships between arbitrary events — be they instantaneous or emergent, atomic or temporally extended. Causal analysis, finally, can make stronger claims about the causal relationship than purely observational analysis by means of intervention; on the other side, however, interventions are inherently restricted to non-emergent, actively forceable manipulations. In cases where relations between complex, emergent events need to be detected, the modeller needs to revert to purely observational analysis, possibly combined with interventions in order to explore particular parts of the parameter space.

Chapter 8

MC²MABS: Monte Carlo Model Checker for Multiagent-Based Simulations

8.1 Introduction

The ideas and algorithms described in the previous chapters have also been implemented into a practical framework as part of the research project. The name of the resulting tool is MC²MABS which stands for ***M**onte **C**arlo **M**odel **C**hecker for **M**ulti**A**gent-**B**ased **S**imulations*. This chapter contains a description of the design and the implementation of MC²MABS, together with a comprehensive empirical performance evaluation.

Following the conceptual ideas described in the previous chapters, MC²MABS is designed as a framework which incorporates the idea of *statistical runtime verification* and whose central characteristic is the *interleaving of simulation and property evaluation*. The framework consists of two central parts: (i) a *simulator* which represents a framework that hosts custom model logic provided in a high-level programming language (C++) and performs the execution of the simulation, and (ii) a *monitor* which implements the algorithms described in Chapter 6 and evaluates a given property upon the traces produced by the simulator.

Communication between the simulator and the monitor happens via a well-defined functional interface which is realised as a set of mandatory and additional optional callback functions to be implemented by the modeller. Apart from the basic logic required by the monitor, the modeller is free to incorporate arbitrary logic into the simulation. In order to offer a good balance between performance and usability, the simulator is implemented in C++.

The monitor is written in Haskell. In order to avoid unnecessary computation, it is based on the idea of *lazy evaluation*, according to which a group state is requested from the simulator if and only if it is strictly required for verification. When requesting a group state, the underlying simulation is triggered to advance for a single step.

The chapter is structured as follows. We first give a general overview of the architecture and design decisions of MC²MABS in Section 8.2. The implementation of the monitor component is described in further detail in Section 8.3. The architecture of the simulator, design decisions and implementation are described in Section 8.4. An example how MC²MABS can be used in a real-world scenario is given in Section 8.5. Finally, the performance of MC²MABS has been analysed in a set of experiments which are described in Section 8.6.

8.2 Overview

As mentioned above, MC²MABS consists of two components: a monitor (written in Haskell) and a simulator (written in C++). Most existing model checkers offer their own model description language. Examples are PRISM's Reactive Modules language [148], SPIN's Promela [128], or MCMAS' ISPL [163]. The main advantage of proprietary model description languages is that they are focussed on relevant aspects of the type of systems that the associated model checker is supposed to verify. For example, the Reactive Modules language provides support for probabilistic state transitions and ISPL provides support for the description of multiagent systems. The logic of the system to be verified can thus be described in a compact way.

Providing a model description language works best if the domain under consideration is well understood and clearly defined. As indicated throughout the previous chapters, this is not the case for agent-based simulations; apart from the large variety of implementation platforms, frameworks, programming languages, etc., there is also no clear consensus about what constitutes an agent-based model at all. Models come in all flavours and may comprise purely reactive agents with a small set of states as well as highly

```
ghc -O2 -fglasgow-exts -prof --make -auto-all -caf-all -fforce-recomp
  -lstdc++ abs.cpp agent.cpp $1 MC2MABS.lhs -o mc2mabs &&
time ./mc2mabs
```

FIGURE 8.1: `mc2mabs.sh` - A startup script for MC^{2}MABS

deliberative agents based on sophisticated cognitive architectures; they may contain probabilistic state transitions, the weight of which may be the result of significantly complex utility calculations. Depending on the problem domain, the environment may also be of arbitrary complexity and represent an aspect of the world in arbitrary detail. Given the variety of models, modelling domains, paradigms and modellers with backgrounds as varying as computer science and philosophy, economics and sociology, geography and military, defining a common model description language will, in our opinion, invariably result in oversimplification and thus severely limit the applicability of the proposed tool.

In order to solve this problem, we decided to replace a modelling language with a more general simulator which serves as a generic umbrella for a wide range different types of models. It is designed as a *service provider interface* which communicates with the monitor via a clearly defined functional interface. The latter comprises a set of callback functions which the user is required to implement in order to embed the custom simulation logic. It is important to note that the simulator cannot only be used to execute newly defined model logic but also to 'replay' the output generated by a different simulation environment. Consider, for example, NetLogo which offers the opportunity to run a simulation multiple times and write the output to a file [240]. Rather than hosting the actual simulation logic (written in NetLogo's own language), the framework described in this chapter could then simply be used to read the NetLogo output file, 'replay' its written simulation traces and pass their content to the monitor for the purpose of approximate verification.

An important design decision was to keep MC^{2}MABS as lightweight as possible. To this end, both components (simulator and monitor) get compiled into a single binary. All the user needs to do is to describe the logic of the simulation to be verified using the simulator by specifying the callback functions as well as arbitrary other functions that may be needed in a single C++ implementation file, for example `model.cpp`. MC^{2}MABS is then compiled and executed by calling a simple Bash script (shown in Figure 8.1) in the Linux terminal as follows:

```
> sh mcmabs.sh model.cpp
```

During the compilation process, the simulator gets compiled into a static library using the default C++ compiler (`g++`). The library is then linked with the monitor which gets itself compiled into a native binary using the Glasgow Haskell Compiler (`ghc`). It is further important to note that communication is unidirectional: the monitor obtains information from the simulator but not vice versa.

Design and development of the monitor and the simulator are described in further detail in Sections 8.3 and 8.4 below.

8.3 The monitor

The monitor is responsible for monitoring the progress of the simulation and evaluating a simLTL property on the simulated traces in an efficient, 'online' way. We omit a detailed description of the code because, thanks to the declarative and purely functional nature of Haskell, the implementation closely resembles the algorithms described in Chapter 6. Rather than being a comprehensive description of the code, the purpose of this section is therefore to emphasise certain design decisions which lead to a more elegant implementation or to better performance of the resulting application.

Apart from being close to the mathematical description of the algorithms, an important decision for choosing Haskell as the underlying programming language was its inherent support for lazy evaluation. Given the potentially considerable complexity of the underlying simulation, unnecessary computation is to be avoided in any case. Against this background, it is, for example, important to postpone calls from the monitor to the underlying simulator (see Section 8.4) until a new group state is strictly required for evaluating the current property (as defined by the expansion laws described in Chapter 6). Furthermore, it is important to keep the underlying simulation strictly *monotonic* (or *forward-oriented*), i.e. such that ticks are simulated in ascending order only and no tick should be simulated twice. Remember that any property ψ gets evaluated on trace fragments which may be smaller than the full trace produced by the simulation. As illustrated in Section 7.5.2, replication is unavoidable in this case. Consider, for example, the following sequence of time steps:

$$l = \langle 1, 2, 3, 4, 5 \rangle \tag{8.1}$$

Now, consider the following set of trace fragments of l of length 2:

$$\mathit{Fragments}(l, 2) = \{\langle 1, 2\rangle, \langle 2, 3\rangle, \langle 3, 4\rangle, \langle 4, 5\rangle\} \tag{8.2}$$

Due to overlapping trace fragments, some time steps appear twice in this case. As the number of ticks $\#l$ increases and the fragment size approaches $\#l/2$, redundancy increases even more. If the monitor were to trigger the underlying simulation in a 'naïve' way by simply calling `Step(t)` every time t appears in a trace fragment, `Step(t)` would be called multiple times and computation would be wasted. This is also problematic since `Step` cannot be expected to be side-effect-free and thus also not reentrant. Calling the function multiple times would therefore result in the creation of multiple different (and erroneous) system states.

In order to solve this problem, Haskell's lazy evaluation strategy is of great help. To illustrate this, consider again the problem of evaluating a property ψ on fragments of a simulation trace t_s using function `CheckTrace` (see Algorithm 21). In an offline setting, t_s would have to be constructed in its entirety prior to evaluation. If ψ is either satisfiable or refutable on a prefix of t_s, computation would be wasted. In order to avoid that, t_s must not be produced prior to evaluation. This was described in Section 6.3 as the basic principle underlying the online model checking algorithm. To this end, instead of holding a sequence of global states (which it would if t_s was the result of a full simulation run), t_s holds a sequence of *thunks*, i.e. not yet evaluated expressions. Instead of holding the simulation result (i.e. the group state) the expressions can be seen as description *how* a group state is to be obtained once it is strictly needed. Thanks to Haskell's lazy evaluation strategy, a thunk is only evaluated if really necessary. In this way, triggering of `Step` in the underlying simulator can be postponed in order to achieve the desired online effect. Furthermore, lazy evaluation differs from normal order evaluation in that arguments are not copied but *shared*. This means that no argument is ever evaluated more than once. In this way, each tick is only simulated once which achieves the strict monotonicity effect mentioned above.

The monitor is implemented in three modules which are briefly described below:

`SimLTL`: This module contains the grammar of simLTL (agent & global layer), as well as functions for translating formulae into PNF (positive normal form), the definition of common logical equivalences as well as pretty printing functions.

ModelChecker: This module contains the definition of the main data structures such as $AState$, $GState$, agent and simulation traces, as well as the major part of the model checking algorithms described in Chapter 6.

MC2MABS: This module represents the main entry point of the monitor. It is responsible for gathering all relevant information from the simulator, providing the infrastructure for and triggering the evaluation process. Most of the collaboration with the simulator happens in this module. As is common in model checking, MC²MABS also provides the possibility to report a counterexample in case of violation. This process is also triggered in this module.

Module `SimLTL` defines a set of types, most notably those for simLTL formulae (see Listing 8.1). They define the grammar according to which both formulae are constructed and parsed in the monitor and how formulae need to be formulated in the simulator. Note that, for performance reasons, the implementation contains elements which are not contained in the grammars described in Chapter 5 (e.g. `ALast`, `AFinally` or `AGlobally`).

Communication between the monitor and the simulator happens via a clearly defined *functional interface*. It is realised technically through Haskell's *Foreign Function Interface (FFI)* which facilitates collaboration between Haskell and C/C++ code. In order to obtain all information necessary for evaluation, the monitor imports and uses the following functions from the simulator as part of the evaluation process:

getProperty(): Returns a correctness property as a string (described further below)

getFormula(id): Returns a simLTL formula associated with a unique configuration id

getFragmentSize(): Returns the currently selected fragment size.

preConf(id): Is called once before configuration id gets started

preRun(id): Is called once before an individual run of configuration id gets started

step(id, t): Performs a single system update step for tick t.

postRun(id): Is called once after an individual run of configuration id has ended

postConf(id): Is called once after configuration id has ended

```
*** 1.1 Agent layer ************************************************

> data AVal = ANumVal Int
>     | AAttribute Int
>     | ANumFunc Int
>     | AVar String
>     | APlus AVal AVal
>     | AMinus AVal AVal
>     | ATimes AVal AVal
>  deriving (Read, Eq)

> data ALTL =
>     ATrue | AFalse | ALast | AAtom Int
>     | ANot ALTL | AAnd ALTL ALTL | AOr ALTL ALTL | AImpl ALTL ALTL | AEquiv ALTL ALTL
>     | AFinally ALTL | ANext ALTL | ASNext ALTL | AGlobally ALTL
>     | AUntil ALTL ALTL | ARelease ALTL ALTL
>     | AVal AVal | AEq AVal AVal | ANEq AVal AVal | AAEq Double AVal AVal |
>     | ANAEq Double AVal AVal | AGt AVal AVal | AGEq AVal AVal | ALt AVal AVal |
>     | ALEq AVal AVal
>       deriving (Read, Eq)

*** 1.2 Global layer ************************************************

> data GVal =
>     GNumVal Int
>     | GNumFunc Int
>     | GVar String
>     | GPlus GVal GVal
>     | GMinus GVal GVal
>     | GTimes GVal GVal
>       deriving (Read, Eq)

> data GLTL =
>     GTrue | GFalse | GLast | GAtom Int
>     | GNot GLTL | GAnd GLTL GLTL | GOr GLTL GLTL | GImpl GLTL GLTL | GEquiv GLTL GLTL
>     | GFinally GLTL | GNext GLTL | GSNext GLTL | GGlobally GLTL
>     | GUntil GLTL GLTL | GRelease GLTL GLTL
>     | GALTL ALTL | GSel ALTL GLTL | GForall GLTL | GExist Int GLTL
>     | GVal GVal | GEq GVal GVal | GNEq GVal GVal | GAEq Double GVal GVal |
>     | GNAEq Double GVal GVal | GGt GVal GVal | GGEq GVal GVal | GLt GVal GVal
>     | GLEq GVal GVal
>       deriving (Read, Eq)
```

LISTING 8.1: The Haskell types for simLTL agent and global layer formulae

At this point, it is useful to briefly clarify the difference between *configurations* and *runs*. A *configuration* can be seen as one particular parametrisation of the underlying simulation model. A configuration consists of a set of *runs*, by means of which the state space is being explored. Model parameters should only be changed between configurations, but not between runs since the latter's sole purpose is to cope with variance and explore the space to a sufficient degree. The usefulness of configurations for the purpose of realising *interventions* is described further below.

As described in the previous chapters, atomic predicates and numeric variables are also represented as external functions which allows for the integration of custom logic into the model checking process. To

1. Parametrisation:

   `getNumAgents/0`: Returns the number of agents

   `getNumTicks/0`: Returns the number of time steps

   `getNumAgentAtts/0`: Returns the number of agent attributes

   `getNumReps/0`: Returns the number of replications

   (**`getFragmentSize/0`**[a]**):** Length of fragment size necessary for evaluation.

2. Property formulation:

   `getProperty/0`: Returns a correctness property as a string

   `getFormula/1`: Returns a simLTL formula associated with a unique configuration id

3. Initialisation and uninitialisation:

   `preConf/1`: Called before the start of a configuration

   `preRun/1`: Called before the start of an individual simulation run

   `postRun/1`: Called after an individual simulation run

   `postConf/1`: Called after the end of a configuration

4. Model logic:

   `step/2`: Performs a system update step

   `gValue/2`: Encapsulates group-level numeric calculations (e.g. aggregations)

   `aValue/3`: Encapsulates agent-level numeric calculations (e.g. aggregations)

   `gPredicate/2`: Encapsulates group-level Boolean evaluations used as atomic predicates in simLTL

   `aPredicate/3`: Encapsulates agent-level Boolean evaluations used atomic predicates in simLTL

[a] Only needs to be implemented if fragments are not maximal, i.e. if their size is different to the number of ticks.

FIGURE 8.2: Functional interface between the monitor and the simulator

this end, in addition to the functions mentioned above, the monitor may also call the following functions during the evaluation process:

`aPredicate(p,a,t)`: Returns the value of the individual predicate p for agent a at tick t.

`gPredicate(p,t)`: Returns the value of the predicate p at tick t.

`aValue(v,a,t)`: Returns the value of attribute v for agent a at tick t.

`gValue(v,t)`: Returns the value of group attribute v at tick t.

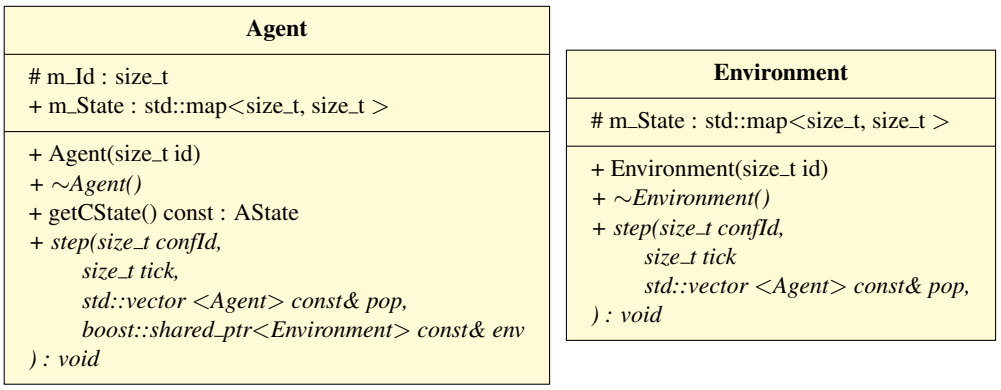


FIGURE 8.3: UML class diagrams of the agent and environment class in the simulator

`aPredicate` and `gPredicate` realise the idea of *agent* and *group predicate functions* formally introduced in Section 5.2.1[1]; analogously, `aValue` and `gValue` realise the idea of *agent* and *group attribute functions*.

An overview of the functional interface is shown in Figure 8.2. The signatures and further descriptions of the internal logic of all functions are given in Section 8.4 below. Since most functions in the simulator operate upon global state and thus cannot be expected to be side-effect-free, their invocation needs to take place within the *IO monad*.

8.4 The simulator

As indicated in Section 8.2, the simulator provides a generic container into which arbitrary simulation logic can be embedded. As mentioned above, a modeller may just want to open existing simulation files that have been produced elsewhere and make their content applicable to the monitor in order to verify it using the approach described in this work.

The design of the simulator largely follows the formal framework of agent-based simulations described in Chapter 3, albeit, for performance reasons, some details have been implemented differently. The logic of the simulator can be subdivided into an *essential* and an *optional* part. The essential part is strictly required by the monitor, the optional part is merely provided for convenience in order to facilitate the implementation of more complex simulations. The framework consists of the following source files:

[1] Note that, in the current implementation, predicates are assumed to be nullary and thus correspond with atomic propositions.

abs.h: Contains declarations of internally used data structures as well as the external and internal interface of the framework. The file does not give any definitions and should not be changed by the user.

abs.cpp: Contains implementations of some core interface functions as well as definitions of additional data structures; it should not be changed by the user.

agent.h: Defines the structure of the agent and the environment class; it should not be changed by the user.

agent.cpp: Implements the constant (i.e. model-independent) logic of the agent and the environment class; it should not be changed by the user.

model.cpp: Contains the model-specific logic which is implemented by the user. The name of the file can also be chosen arbitrarily. It is passed to the compilation script as described in the previous section.

For performance reasons, the framework is largely implemented in a procedural, non-object-oriented way. The only two classes are `Agent` and `Environment`, both of which are part of the optional, convenience-based fragment of the framework. UML class diagrams are given in Figure 8.3. Note that an agent's internal data structures are declared *public* and are thus accessible to everyone from outside which, strictly speaking, violates the principle of encapsulation. This has been done for reasons of performance since it allows agents to access the state of other agents directly, without further indirection. It is also important to note that the data structures that store an agent's internal state assume that all attribute values are numeric. This has also been done deliberately for performance reasons. First, numeric values can generally be processed more efficiently than, for example, text strings. Secondly, information exchange between the simulator (C++) and the monitor (Haskell) requires marshalling of the underlying data types which can be costly — especially for list-based data types such as strings. In order to make numeric representations of attribute values more maintainable, enumerations can be used (for an example, see Section 8.5).

The global function `step` (which is called by the monitor to obtain a new group state) represents a central and essential part of the interface. It first calls function `step` on the environment object as well as the equally named function on each agent instance. Both functions are model-specific and need to be implemented by the user as part of the model description process (an example implementation is

given in Section 8.5). After all updates have been performed, an array which represents a group state is created and returned to the monitor.

The user may decide not to use the agent and environment classes and instead provide a different implementation. In this case, he only needs to make sure that the value returned by function `step` (the group state) has the right format.

The simulator further defines a number of global data structures which are briefly described below. The population of agents is stored as a vector of `Agent` instances. The environment is stored as an instance of `Env`. In order for the global state of the model to be easily accessible to the monitor, it is stored in a central data structure `gtrace` which represents a dynamic array of dynamic arrays of agent states and thus reflects the structure described in Chapter 3. The framework contains a default random number generator which can be accessed from everywhere and can, for example, be used to model random agent decisions. The framework also contains a number of helper variables which are not described here.

In order to describe information exchange between the simulator and the monitor, it is important to briefly revisit the notion of *interventions* described in Section 7.6.2. Remember that an intervention is defined as the intentional manipulation of certain parameters' values. In MC^{2}MABS, interventions are practically realised by means of *configurations*, each of which represents a certain parametrisation of the underlying model. In order to avoid the difficulties resulting from the large variety of possible interventions indicated above and in order to offer the user the highest level of flexibility, we decided not to integrate an explicit intervention mechanism in MC^{2}MABS. Instead, users can create different configurations of the same model (each of which represents a particular intervention) and refer to them with their respective unique id. As described in Section 6.4, function `Estimate` accepts a configuration id as its first argument which it passes on to the `PreConf` and `PostConf` functions of the underlying simulation in order to make it aware of which configuration is currently active. Changes of arbitrary complexity can then be realised by the user in a flexible way using C++ as a higher-level programming language.

A configuration is uniquely described as a textual string with the following structure:

$$Pr(id, fragmentSize, numAgents, numAtts, numTicks, numReps)$$

where id is the configuration id, $fragmentSize$ is the length of trace fragments necessary, $numAgents$ is the number of agents, $numAtts$ is the number of agent attributes, $numTicks$ is the number of time

```
1 > data Prop = Atom Conf
2 >           | Div Prop Prop
3 >           | Mult Prop Prop
4 >           | Add Prop Prop
5 >           | Sub Prop Prop
6 >         deriving (Eq, Read, Show)
```

LISTING 8.2: The correctness property type

steps to be simulated and *numReps* is the number of replications to be evaluated. An example of a configuration string is given below:

```
Pr(1, 2, 100, 3, 100, 1000)
```

A configuration represents one particular parametrisation of the underlying simulation and is uniquely identified by its id. Each configuration is associated with one particular simLTL formula to be checked (the actual property). It is important to note that the property is not part of the configuration string but returned by function `getFormula` whenever it is needed by the monitor. The evaluation of a particular configuration results in a probability value (denoted by `Pr` in the configuration string).

The configuration string is assembled by function `getConf` which itself calls several internal functions in order to collect the necessary information. Those internal functions that need to be implemented by the user as part of the model description process are shown below:

`getNumAgents():` Returns the number of agents in the model.

`getNumTicks():` Returns the number of time steps to be simulated.

`getNumAgentAtts():` Returns the number of attributes per agent.

`getFragmentSize():` Returns the required trace fragment size. The default value is `getNumTicks()`.

The starting point for verification is a *correctness property*. As described in Chapter 7, some types of validation (e.g. conditional or causal analysis) require the evaluation of multiple individual simLTL formulae and can thus be considered 'higher-order properties'. To this end, a correctness property is defined as a recursive type, as shown in Listing 8.2.

According to that type definition, a correctness property is either a single atomic property or a simple arithmetic expression formulated over two correctness properties. In order to exemplify that, consider

1. Initialisation and uninitialisation:

   _preConf/1: Custom logic to be executed before the start of a configuration

   _preRun/2: Custom logic to be executed before the start of a run

   _preTick/3: Custom logic to be executed in the beginning of a tick

   _postTick/3: Custom logic to be executed in the end of a tick

   _postRun/2: Custom logic to be executed in the end of a run

   _postConf/1: Custom logic to be executed in the end of a configuration

2. Model logic:

   Agent:step/4: Custom logic to be performed as part of a single agent update

   Environment:step/3: Custom logic to be performed as part of the environment's update

FIGURE 8.4: Functional hooks provided by the simulator for convenience

again the intervention example given in Section 7.6.2. Remember that we want to find out whether the infection status of a particular agent has an impact on the overall infection status of the population. We assume the existence of two configurations with ids 1 and 2, in the first of which the particular agent is infected, in the second of which the agent is susceptible. Both configurations use the same simLTL formula and the same configuration data (1,000 agents, 100 ticks, 100 replications, 1 agent attribute, fragment length 100). We can thus formulate the following property:

```
Sub (Atom Pr(1, 100, 100, 1, 100, 1000)) (Atom Pr(2, 100, 100, 1, 100, 1000))
```

This expressions causes the monitor to perform two evaluations — one for each atomic property — and to subtract their results in order to obtain the overall probability. It is now up to the user to check whether the resulting value is less than or equal to the desired threshold.

The functions that were shown in Figure 8.2 further above are strictly necessary, i.e. their existence is strictly required by the monitor. They provide a basic skeleton into which custom model logic can be embedded. As long as compliance with the functional interface is guaranteed, users are free to structure their implementation around those functions in whatever way they prefer. In order to provide a higher level of convenience, however, some of the functions in Figure 8.2 have been given default implementations. For example, function `getFragmentSize` returns the total length of the path by default. If a different fragment size is necessary, the content of the function can be adapted accordingly. In order to separate the framework code from the user code, some default implementations (most of which can be found in file `abs.cpp`) of particular functions forward the call to *custom versions* of the

```
1  void postRun(size_t confId)
2  {
3    _postRun(confId, population);
4
5    // iterate over ticks
6    for(int i=0; i<getNumTicks(); i++)
7    {
8      if(gtrace[i] == NULL)
9        continue;
10
11     // iterate over agents
12     for(size_t j=0; j<getNumAgents(); j++)
13     {
14       // iterate over attributes
15       for(size_t k=0; k<getNumAgentAtts(); k++)
16       delete[] gtrace[i][j][k];
17       delete[] gtrace[i][j];
18     }
19     delete[] gtrace[i];
20   }
21   delete[] gtrace;
22   population.clear();
23 }
```

LISTING 8.3: Function `postRun`

same or additional functions which are not part of the interface between the monitor and the framework but serve as *hooks* which allow for the implementation of custom logic by the user. The reason for that is to separate generic boilerplate code which is common to all models from model-specific logic provided by the user. A custom version of a function has the same name as the original function, preceded with an underscore. Consider, for example, function `postRun` which is shown in Listing 8.3. The function is called after a run has completed; its purpose is to clean up memory allocated for that particular run and reset the population to its initial state in order for simulation runs to be entirely independent. Nevertheless, there might be additional cleanup logic that the user wants to execute. To this end, function `postRun` calls function `_postRun` which can be implemented by the user (see Line 3). An overview of all functional hooks provided for convenience by the simulator is shown in Figure 8.4.

Apart from compliance with the functional interface shown in Figure 8.2, the user is free to add any additional logic in the form of C++ code wherever necessary. The only restriction which is important (but currently not enforced by the framework) is that functions `gValue`, `aValue`, `aPredicate` and `bPredicate` have to be side-effect-free since they are called from purely functional code. This is in contrast to all other callback functions which are called from within the IO monad and are thus allowed to have side effects. We intend to enforce the purity of those four functions by the simulator in the future.

```
size_t getNumAgents() { return 1000; }
size_t getNumTicks() { return 100; }
size_t getNumAgentAtts() { return 1; }
size_t getNumReps() { return 100; }
size_t getFragmentSize() { return 2; }
```

LISTING 8.4: The parametrisation functions

```
void _preConf(int confId)
{
  // setup neighbourhood relationship
  for(int i=0; i<getNumAgents(); i++)
  {
    // create 5 random neighbours per agent
    for(int j=0; j<5; j++)
      nb[i].push_back(rand() % 1000);
  }
}
```

LISTING 8.5: Implementation of function `_preConf`

8.5 Example

This section illustrates the usage of MC²MABS by describing the implementation of a simple real-world model and the verification of a simple temporal property. The example has been deliberately kept brief in order to focus on the general implementation of a model rather than on the model logic itself. A more realistic and complex example with multiple properties is presented in Chapter 9.

Various variants of a simple disease transmission model have been used as the running example throughout the previous chapters, it shall also be used as an example here. A description of the agents' behavioural rules was given in Section 5.1.1. In order to implement this model in MC²MABS, we first need to think about the configuration parameters. We assume that the simulation comprises 1,000 agents, runs for 100 ticks and evaluation requires 100 replications. Each agent comprises a single attribute *health* which can be either *susceptible*, *infected* or *recovered*. We can thus implement the parametrisation functions as shown in Listing 8.4. Note that the fragment size is set to 2. The reason for that is given further below.

We further assume that agents are connected to each other in a certain way which defines their neighbourhood relationship. The topology is assumed to be constant for a single configuration, we can therefore use function `preConf` to implement the desired logic. Remember that `preConf` is a core function which also contains model-independent logic and should thus not be modified by the user. In order to inject model-specific logic, function `_preConf` can be used instead. We construct in this

```
void _preRun(size_t confId, vector<Agent>& population)
{
  // initialise health state of agents
    int idx=0;
  vector<Agent>::iterator it;
  for(it=population.begin(); it!=population.end(); ++it) {
    if(idx++ == 0)
      it->m_State[HEALTH] = INFECTED;
    else
      it->m_State[HEALTH] = SUSCEPTIBLE;
  }
}
```

LISTING 8.6: Implementation of function `_preRun`

```
enum Attribute
{
  HEALTH=0
};

enum Health
{
  SUSCEPTIBLE=0,
  INFECTED=1,
  RECOVERED=2
};
```

LISTING 8.7: Two helper enumerations describing an agent attribute and its valuation

function a random network of agents and store the topology in a hash map variable called `nb`. The implementation of `_preConf` is shown in Listing 8.5. Note that the algorithm is not optimal since it allows for self-loops and duplicate links, the latter of which may cause the actual number of neighbours to fall below 5. For the sake of simplicity, however, we accept these deficiencies here.

The properties to be verified upon the transmission simulation are expected to be evaluated on multiple replications. In order to avoid ordering effects, we need to make sure that each run operates on the same starting state. To this end, we need to initialise agents appropriately. This needs to be done on a run-per-run basis and the logic is thus implemented in the `_preRun` function (see Listing 8.6).

We also need to think about uninitialisation. All we need to make sure in this case is that the neighbourhood relationship, which has been constructed for an individual configuration, is deleted after the configuration has finished. To this end, we need to call `nb.clear()` in function `_postConf`.

The next part of the implementation concerns the behaviour logic of the agents and the environment. As described above, all attribute values are required to be numeric for performance reasons. In order to deal with numeric values in a convenient way, we add two enumerations as shown in Listing 8.7. Based on that, the logic of the agents and the environment can now be implemented. In the disease transmission

```
void Agent::step(
  size_t confId,
  size_t tick,
  vector<Agent> const& population,
  boost::shared_ptr<Environment> const& env
)
{
  if(m_State[HEALTH] == SUSCEPTIBLE)
  {
    // determine number of infected neighbours
    float cnt=0;
    vector<int>::iterator it;
    for(it=nb[m_Id].begin(); it!=nb[m_Id].end(); ++it) {
      StateMap::const_iterator itState = population[*it].m_State.find(HEALTH);
      if(itState->second == INFECTED)
        cnt++;
    }
    // transition to infected based on number of infected neighbours
    if(getRand() <= (cnt/5.0)) m_State[HEALTH] = INFECTED;
  }
  else if(m_State[HEALTH] == INFECTED) {
    if(getRand() <= 0.7) m_State[HEALTH] = RECOVERED;
  }
}
```

LISTING 8.8: Example implementation of an agent's step function in the transmission model

example, there is no environment and we can thus leave the implementation of `Environment:step` empty. The agent class needs to implement the protocol shown in Algorithm 2 in Section 5.1.1. An example implementation is shown in Listing 8.8.

This concludes the implementation of the simulation. We now need to think about the properties that we want to check. The behavioural protocol states that, once an agent has recovered, it will always remain so. This is, for example, expressed by Property 7.22 which was given in Section 7.5.1.1. A slightly modified, universally quantified, version is shown below.

$$\forall((health = recovered) \overset{x}{\rightsquigarrow} (health = recovered)) \tag{8.3}$$

The property is *exhaustive* in such a way that it makes a statement about *all* agents in the population in *every* time step. Its evaluation is thus fairly complex. Alternatively, one could formulate the question as a property about a single agent's probability to exhibit the desired behaviour. Since, in our example, agents are all following the same behavioural protocol, this sampling approach would work well. We can thus formulate the following property:

$$(health = recovered) \overset{x}{\rightsquigarrow} (health = recovered) \tag{8.4}$$

```
const char* getFormula(int idx)
{
  string f = "GALTL (AImpl (AEq (AAttribute 2) (ANumVal 2))
                (AWNext (AEq (AAttribute 2) (ANumVal 2))))";
  return f.c_str();
}
```

LISTING 8.9: Function `getFormula` which returns the simLTL formula to be evaluated

```
const char* getProperty()
{
  string conf = getConf(
    1,
    getCheckingMode(),
    getFragmentSize(),
    getNumAgents(),
    getNumAgentAtts(),
    getNumTicks(),
    getNumReps()
  );
  return conf.c_str();
}
```

LISTING 8.10: Function `getProperty` which constructs and returns the final configuration string

As indicated above, simLTL formulae are represented in MC²MABS as instances of the types shown in Figure 8.1. In order for our properties to be checkable, we thus need to reformulate them according to this grammar and return them in function `getFormula`. An example implementation for Property 8.4 is shown in Listing 8.9.

Note that, since all simLTL properties being evaluated have to be full simLTL formulae (i.e. group formulae), we need to use `GALTL` to denote that the subsequent formula is to be checked on a single agent. Since the `GALTL` formula is not wrapped into a quantifier, it will be evaluated on a randomly chosen agent trace (as described in the discussion of Function `CheckAgentFormula` in Section 6.3.3) which achieves the desired effect of random sampling. It should also be clear now why the fragment size has been set to 2 in Listing 8.4. Instead of formulating the property as a temporal invariant using 'globally', we formulated it as a simple property about state pairs. In order to obtain the correct result, it thus needs to be checked on fragments of size 2.

Function `getProperty` which assembles and returns the final configuration string can now be formulated as shown in Listing 8.10. If we assume that the model is implemented in file `transmission.cpp`, then MC²MABS can be invoked from the command line as follows:

```
> sh mcmabs.sh transmission.cpp
```

This produces the following output which shows the evaluation result for each replication, the averaged overall result for each configuration (in this case just one) and the overall result of the entire property:

```
Checking configuration 1
Checking 100 replication(s) ...
Rep. 1; result = Just 1.0;
Rep. 2; result = Just 1.0;
Rep. 3; result = Just 1.0;
Rep. 4; result = Just 1.0;
Rep. 5; result = Just 1.0;
.
.
.
Rep. 98; result = Just 1.0;
Rep. 99; result = Just 1.0;
Rep. 100; result = Just 1.0;
Configuration result: 100.000%
RESULT: 100.000%
```

This indicates that our property is, in fact, true and agents comply with the rule that, once they have recovered, they will not become infected again. This concludes the implementation of the example model.

8.6 Evaluation

This section describes the evaluation of MC²MABS against the background of both time and memory consumption. The following experiments have been performed:

1. Impact of *population size* on execution time and memory consumption (for both unquantified and universally quantified formulae)
2. Impact of *formula size* on execution time and memory consumption (for both unquantified and universally quantified formulae)

3. Impact of *fragment size* on execution time and memory consumption
4. Impact of *satisfiability/refutability* on execution time

Since the executable binary file of MC^{2}MABS is a merge of code written in both C++ and Haskell, their independent evaluation is not easily possible. For that reason, all individual measurements had to be derived from the profiling results of the entire application. We focus on five major tasks:

Simulation (SIM): Time spent on executing the model logic; this describes the performance of the simulator written in C++

Extraction (EXT): Time spent on extracting and transforming (*marshalling*) the traces created by the C++ simulator into their corresponding Haskell vectors

Verification (VER): Time spent on actual verification, i.e. evaluation of the simLTL property

Other (OTH): Time spent on 'housekeeping', i.e. other, non simulation- or evaluation-related tasks such as garbage collection, system calls and profiling itself

Total (TOT): Total runtime of MC^{2}MABS

The evaluation was performed using `gprof`, Haskell's built-in profiling system on a 64 Bit Dell Latitude Laptop with two Intel® Core™ 2 Duo CPUs (2.8 GHz each), 8GB of memory and Linux Mint Olivia (kernel version 3.8.0-35) as operating system. All results are based on experiments involving 100 replications of the underlying simulation. In order to obtain the relevant profiling information, MC^{2}MABS was compiled with the following command line arguments:

```
ghc -O2 -L. -fglasgow-exts -prof --make -auto-all -caf-all \\
  -fforce-recomp -lstdc++ -rtsopts abs.cpp agent.cpp \\
  ./models/evaluation.cpp MC2MABS.lhs -o mc2mabs
```

An overview of the compilation parameters and their functionality can be found in the Haskell documentation online[2]. `evaluation.cpp` refers to the file containing the model used during evaluation. It is a simple version of the transmission simulation used as a running example throughout this thesis. The code of the agent update logic is shown in Listing 8.11. `prof_mcmabs.sh` refers to a shell script

[2] http://www.haskell.org/ghc/docs/7.6.3/html/users_guide/profiling.html (accessed: 05/14)

```
void Agent::step(
  size_t confId,
  size_t tick,
  vector<Agent> const& population,
  boost::shared_ptr<Environment> const& env
)
{
  if(m_State[HEALTH] == SUSCEPTIBLE) {
    if(getRand() <= 0.3)
      m_State[HEALTH] = INFECTED;
  }
  else if(m_State[HEALTH] == INFECTED) {
    if(getRand() <= 0.5)
      m_State[HEALTH] = RECOVERED;
    else
      m_State[HEALTH] = INFECTED;
  }
  else
  {
    if(getRand() <= 0.7)
      m_State[HEALTH] = SUSCEPTIBLE;
    else
      m_State[HEALTH] = RECOVERED;
  }
}
```

LISTING 8.11: Agent update logic for the evaluation model

which performs a timed execution of MC²MABS and converts the heap profiling output into a graphical representation using `hp2ps`:

```
time ./mc2mabs +RTS -pa -h && hp2ps -e8in -c mc2mabs.hp
```

8.6.1 Runtime

The first set of experiments concerns the impact of various factors on the overall runtime of MC²MABS as well as on the runtime of the individual subtasks mentioned above.

8.6.1.1 Impact of population size

The first experiment evaluates the impact of the population size on the performance of MC²MABS for simLTL formulae of different complexity; the results are shown in Figure 8.5. The lines with slope ≈ 1 on the log-log plots indicate that MC²MABS scales linearly with the size of the underlying population for simulation, extraction and verification. The largest growth in time consumption is shown by other tasks

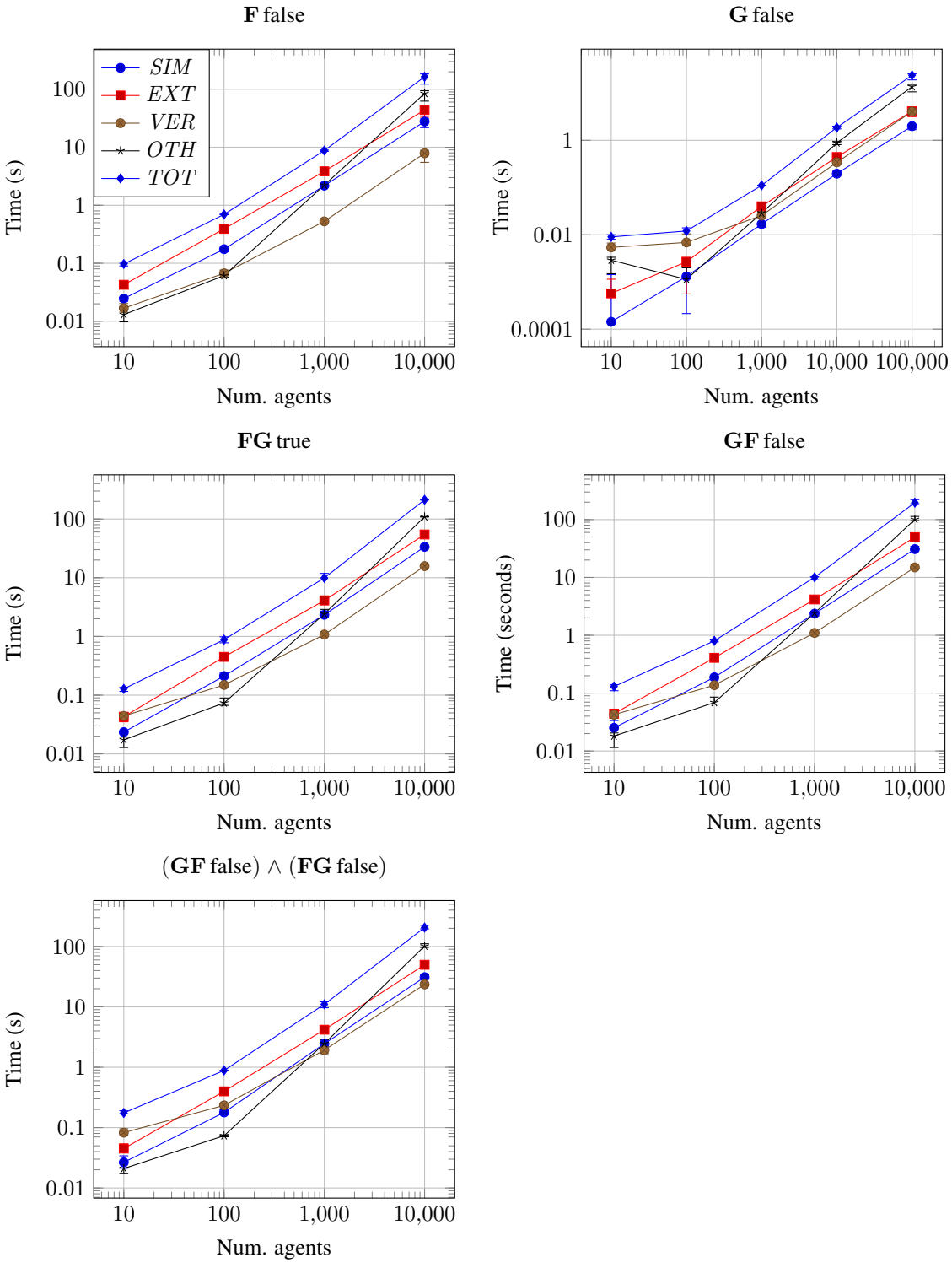


FIGURE 8.5: Influence of population size on total time consumption for unquantified formulae (log-log scale)

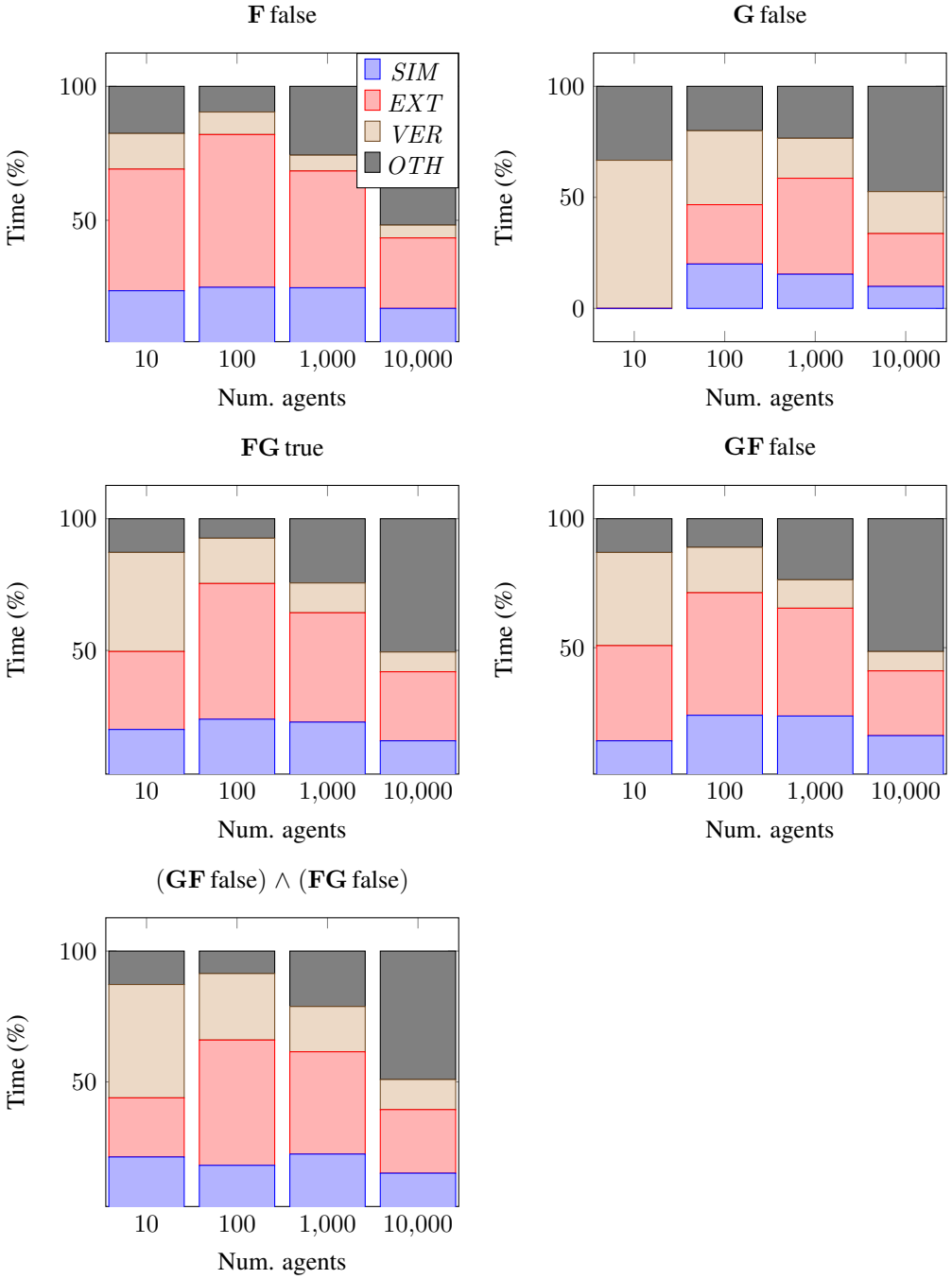


FIGURE 8.6: Influence of population size on relative time consumption for unquantified formulae

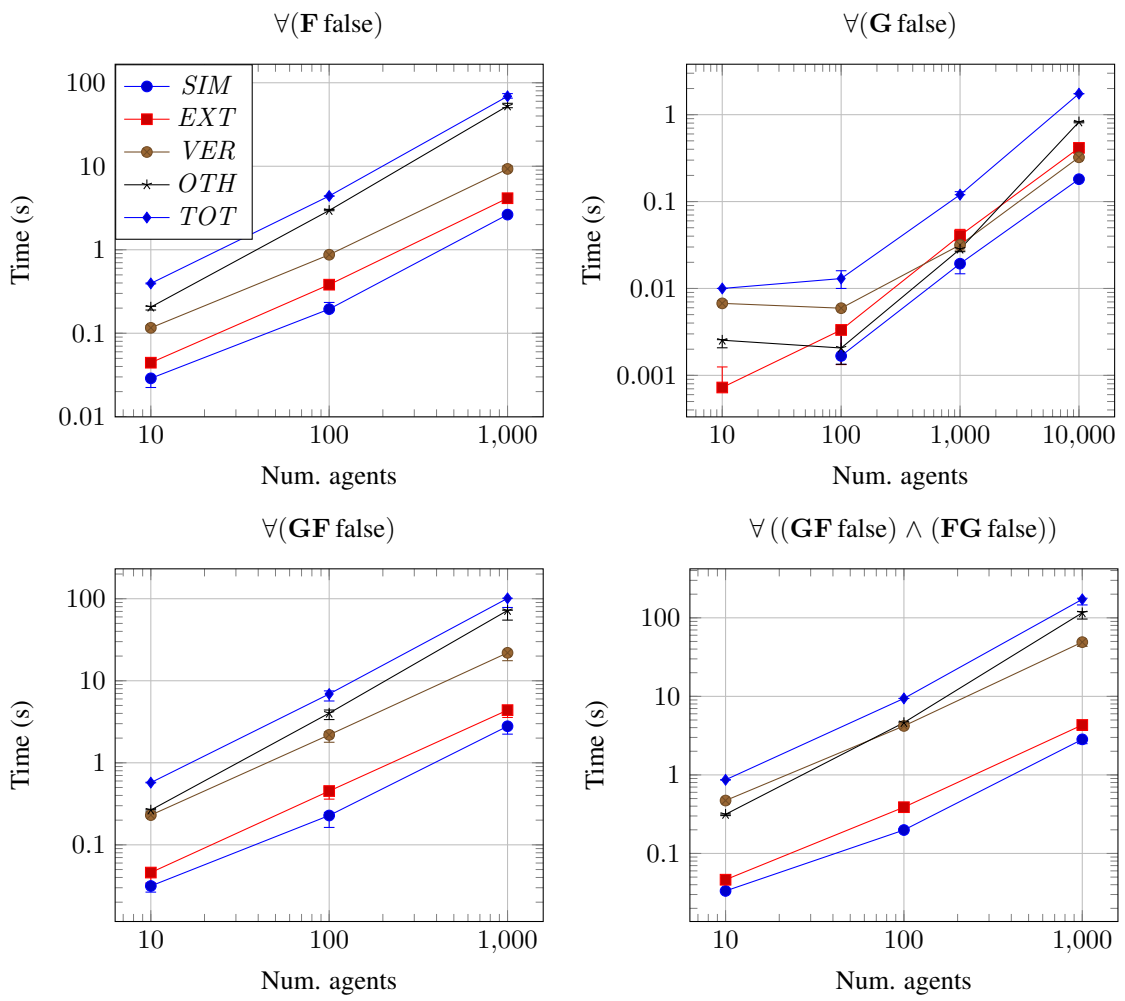


FIGURE 8.7: Influence of population size on total time consumption for universally quantified formulae (log-log scale)

(*OTH*), especially garbage collection. This becomes particularly apparent as the number of agents grows; in the case of 10,000 agents, the time spent on housekeeping exceeds that of the other tasks.

The percentage of time spent on each of the four steps is illustrated in Figure 8.6. It becomes apparent that, as the population size grows, an increasing amount of time is spent on extraction (i.e. marshalling the data structures between C++ and Haskell) and housekeeping, in particular garbage collection. This may represent a bottleneck for large populations which we aim to address as part of the future work.

The numbers also show that the 'satisfiability' of the formula has a large impact on the time spent on each task. Consider, for example, formula '**G** false' shown in the upper right plot of Figure 8.7. The

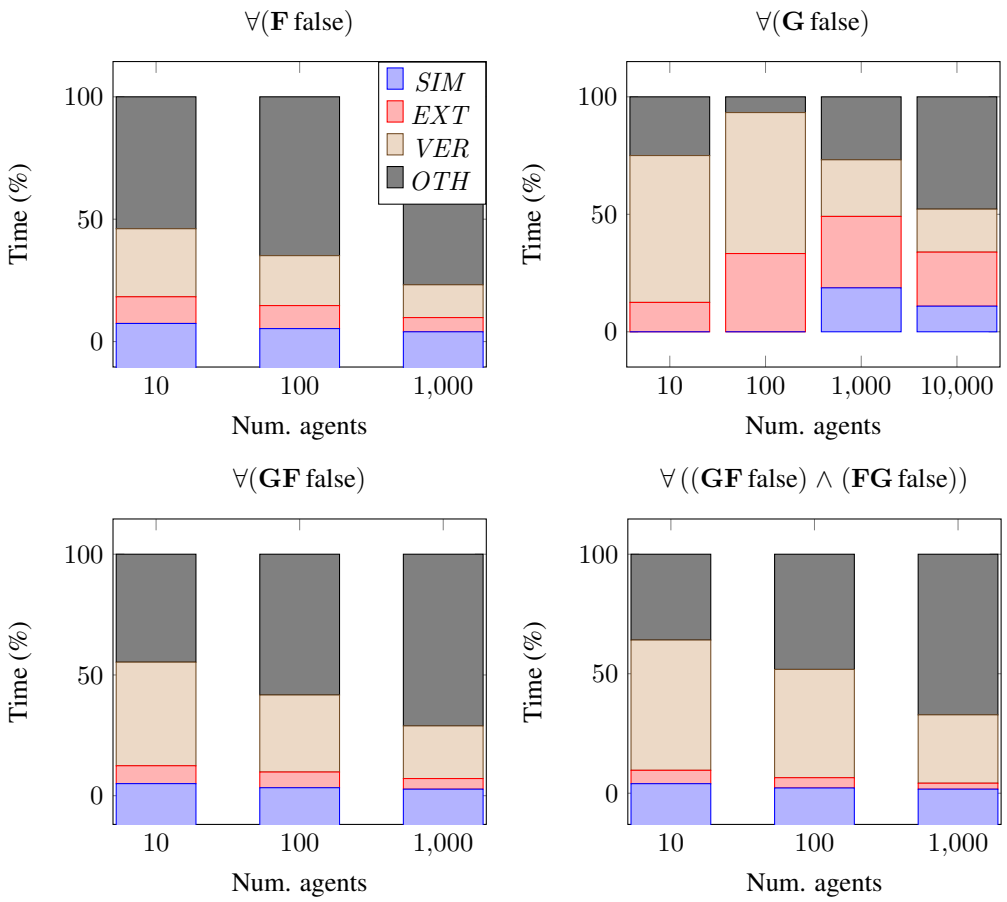


FIGURE 8.8: Influence of population size on relative time consumption for universally quantified formulae

formula is immediately refutable; as a consequence, evaluation is very quick, even for large populations of 100,000 agents. The impact of satisfiability/refutability is analysed in more detail in a separate experiment (see Section 8.6.1.4).

A second set of experiments which investigates the impact of population size has also been conducted for universally quantified formulae; the results are shown in Figure 8.7 (absolute numbers) and 8.8 (relative numbers). MC²MABS also scales linearly in this case. As shown in Figure 8.8, however, housekeeping (particularly garbage collection) becomes a serious overhead as the number of agents grows. We plan to address this issue in the future, e.g. by employing strictness in some of the operations.

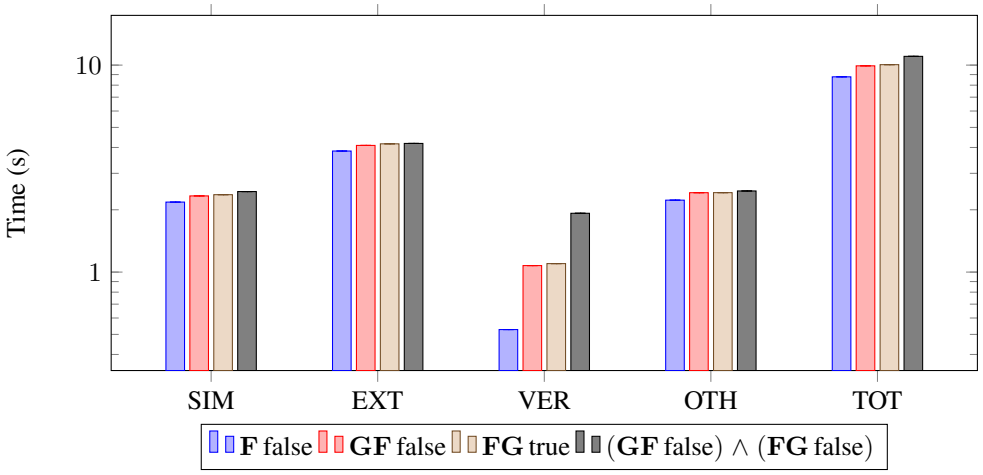


FIGURE 8.9: Influence of formula size on evaluation time for unquantified formulae

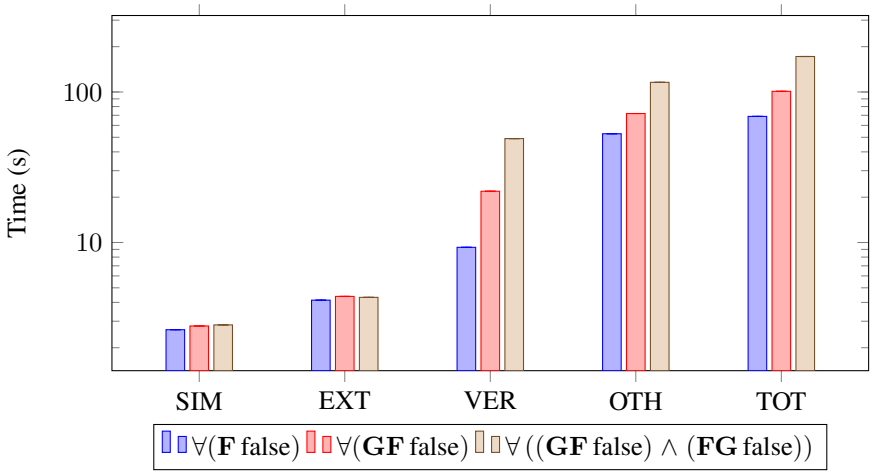


FIGURE 8.10: Influence of formula size on evaluation time for universally quantified formulae

8.6.1.2 Impact of formula size

The second experiment concerns the impact of the formula size on the overall runtime performance of MC²MABS; the results are shown in Figure 8.9. The numbers indicate that the time spent on simulation, extraction and other tasks is independent of the size of the formula. As expected, however, formula size has a clear impact on verification speed.

Similar to the previous experiment, the impact of formula size has also been determined for universally quantified formulae; the results are shown in Figure 8.10. In addition to verification, formula size also has an influence on other tasks, most notably housekeeping.

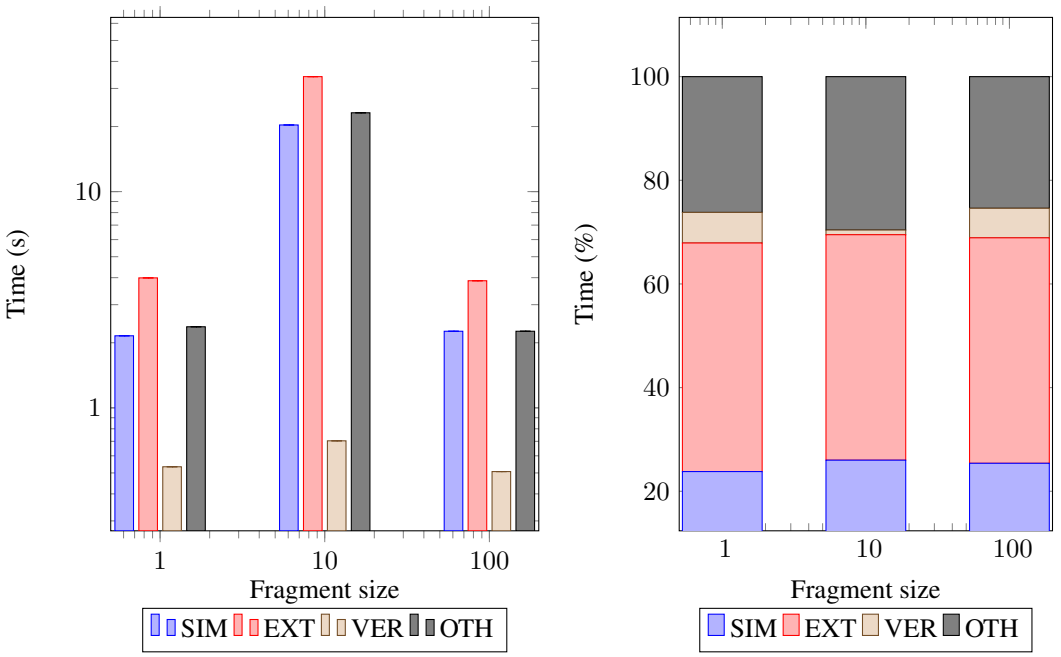


FIGURE 8.11: Influence of fragment size on absolute (left) and relative (right) evaluation time

8.6.1.3 Impact of fragment size

Since several formulae require evaluation on trace fragments of different length, the impact of the fragment size on the overall performance of MC²MABS has also been evaluated. For this experiment, formula '**G** true' has been evaluated on a population of 1,000 agents for fragment sizes of 1 (single states), 10 and 100 (the full trace). The results are shown in Figure 8.11. The numbers illustrate that, in terms of total runtime, the fragment size has a 'bell-shaped' impact on all steps of the evaluation which can be explained as follows. In case of fragment size 1, each 'real' state is represented only once, as is the case for fragment size 100 (only if the overall number of ticks is also 100, of course). In the case of fragment size 10, however, each state may be part of several trace fragments; states are therefore duplicated which causes an increase in time consumption for all steps. The numbers in Figure 8.11 (right) also indicate that, in the case of fragment size 10, the amount of time spent on other tasks is more and the amount of time spent on verification is less than in the case of fragment size 1 or 100.

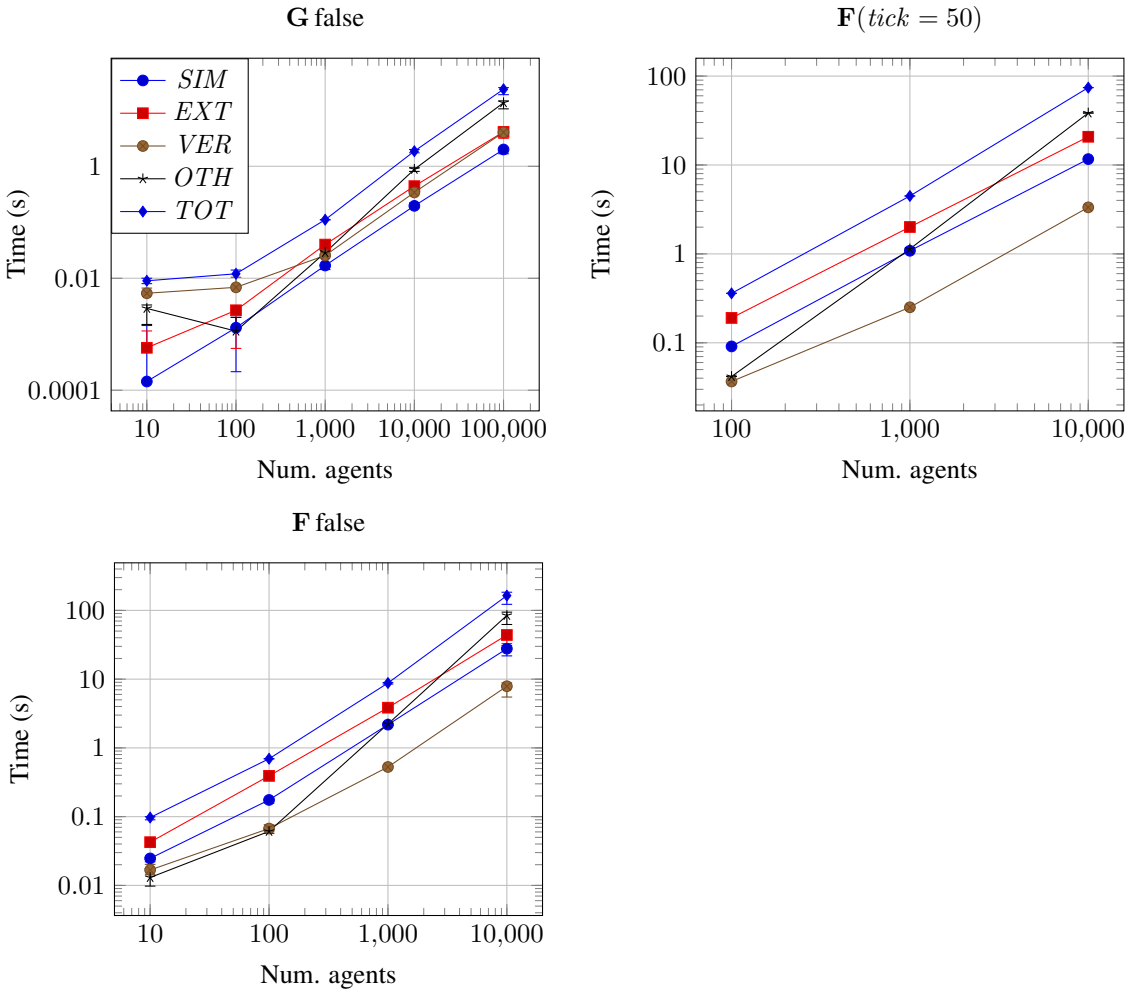


FIGURE 8.12: Impact of satisfiability/refutability on runtime for unquantified formulae (log-log scale)

8.6.1.4 Impact of satisfiability/refutability

The final experiment in the runtime section concerns the impact of satisfiability/refutability. The goal of this dissertation project is to develop a runtime verification approach which reports success or failure as soon as possible; to this end, we expect formulae which can be satisfied or refuted early to be evaluated in less time than formulae which require analysis of the entire trace. In order to test this hypothesis, we checked the evaluation of three properties and their impact on runtime:

'G false': can be refuted immediately in the first state

'F$(tick = 50)$**':** can be refuted in state 50, i.e. after 50%, and

'F false': cannot be refuted until the end, i.e. requires examination of all states

The results are shown in Figure 8.12. The numbers indicate that the impact of satisfiability/refutability is as expected: for formulae which can be refuted early, even large populations can be analysed in a small amount of time.

8.6.2 Memory allocation

Profiling memory consumption for a lazy language like Haskell can be difficult. For example, expressions without arguments, so-called *Constant Application Forms (CAFs)*, are evaluated only once and shared for later use. Due to their global scope, CAFs are thus, strictly speaking, not part of the call graph and hence need to be treated differently. Straightforward analysis of memory allocated within the call graph only can thus be misleading. In the analysis below, all CAFs are aggregated under the 'Other' section.

8.6.2.1 Impact of population size

For simplicity, we restrict the analysis of memory consumption to the case of unquantified formulae; the results are shown in Figure 8.13. The numbers indicate that memory consumption for both verification and simulation is constant and memory consumption for extraction and marshalling increases linearly with the population size. The overhead of extraction/marshalling becomes even more obvious when we consider the relative memory allocation per evaluation step as shown in Figure 8.14.

It is important to note, however, that total memory allocation is not sufficient for understanding the full allocation behaviour of the program. It is useful to also analyse the *runtime heap profile* which describes memory allocation *over time*. The profiles for the first experiment (using property '**F** false') are given in Figures 8.15-8.17. Due to the extremely short runtime, the heap profile for a population of 10 agents is subject to considerable noise which limits its informative value; it has thus been omitted from the analysis. For clarity, we also restrict the number of functions and data structures to 10. In the graphs, 'Pinned objects' refer to information in memory which is not movable by the garbage collection, for example memory allocated in the C++ part of the application. Furthermore, some of the functions described in Chapter 6 are split up into two separate functions (one *outer* and one *inner* part of the

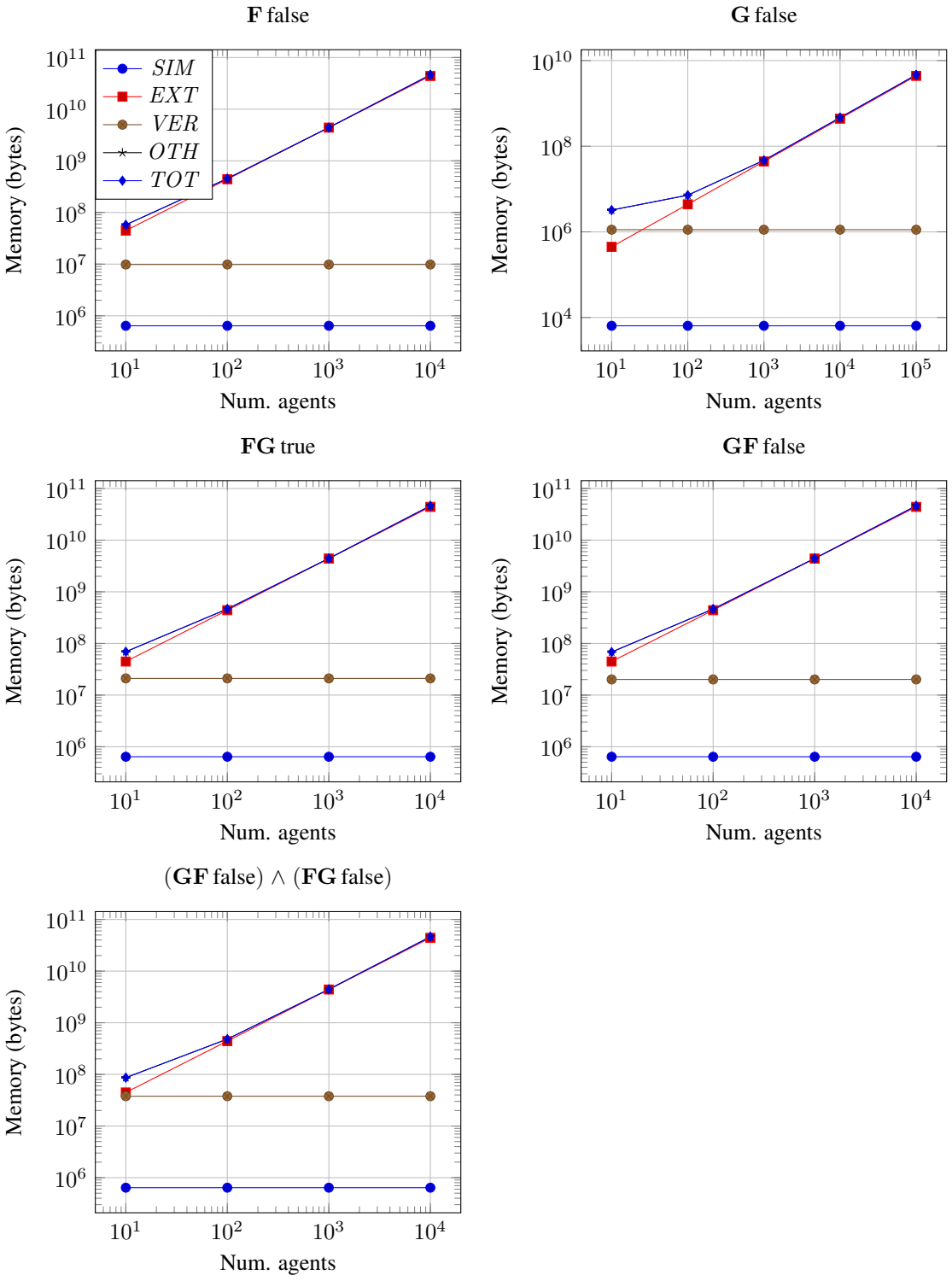


FIGURE 8.13: Absolute memory allocation for unquantified formulae (log-log scale)

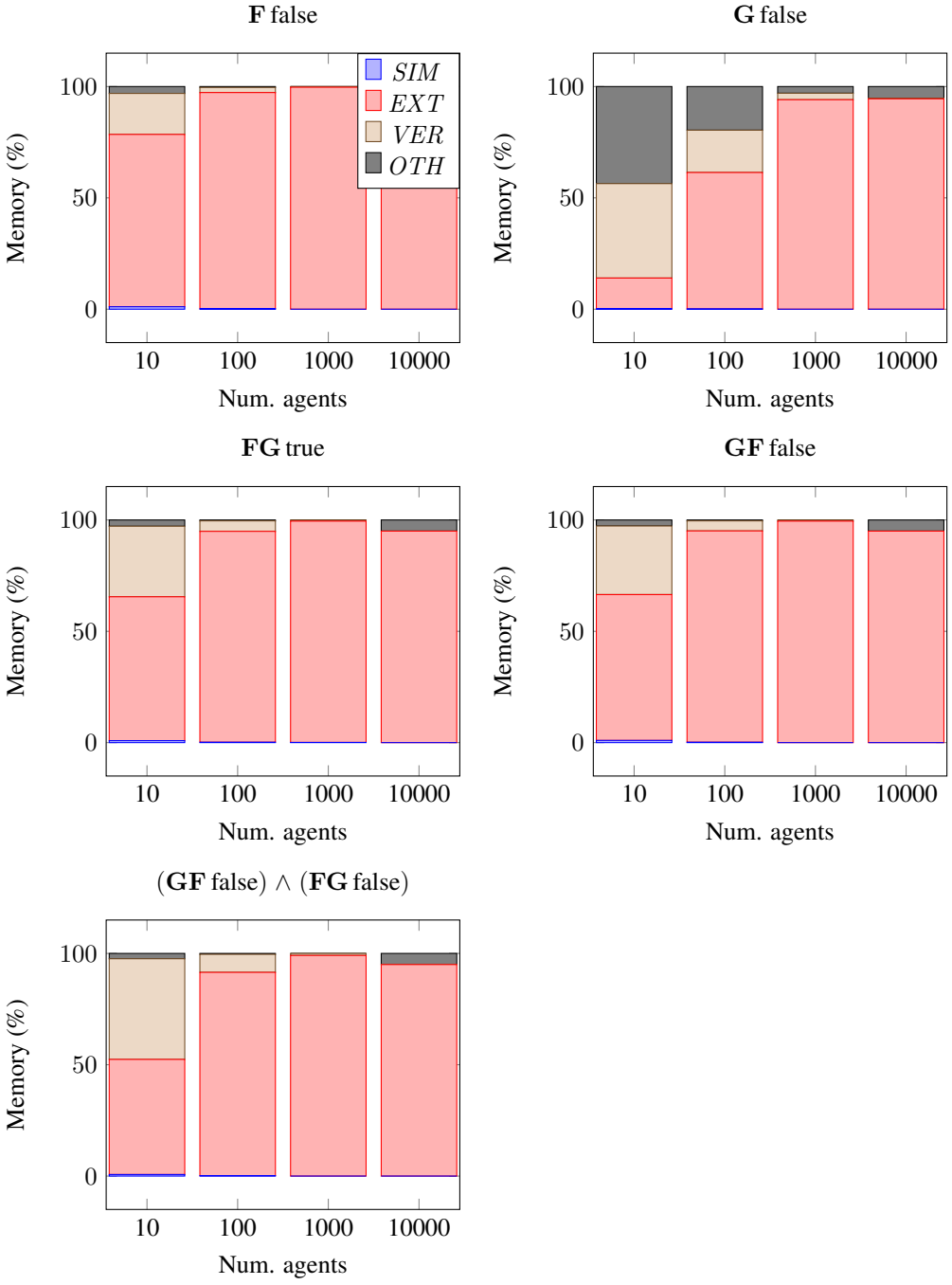


FIGURE 8.14: Relative memory allocation for unquantified formulae

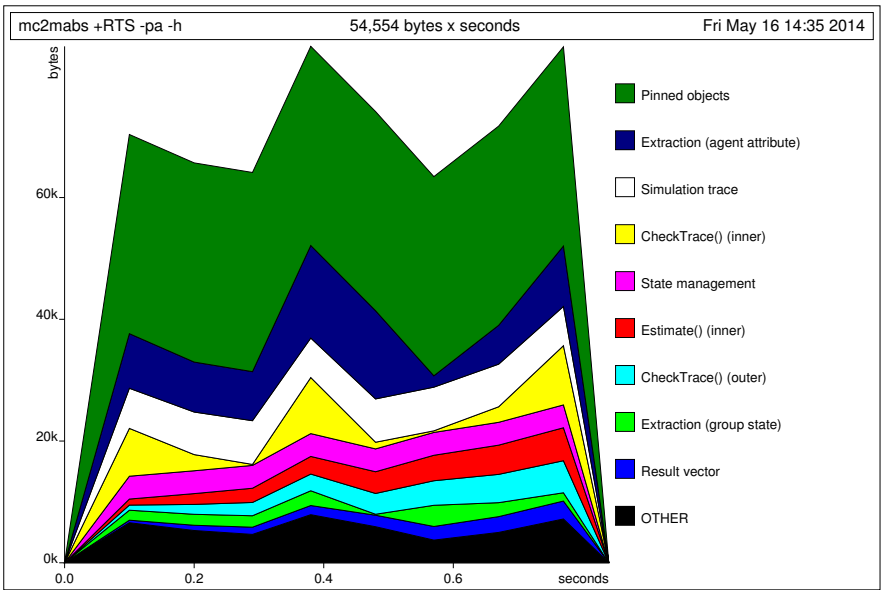


FIGURE 8.15: Runtime heap profile for 100 agents

overall function) in the Haskell implementation for technical reasons; this distinction is also reflected in the graphs. Finally, due to the pure functional nature of Haskell, global state cannot be maintained. In order to emulate this functionality, alternative options such as the *State Monad* have to be used. This state management accounts for a certain level of memory allocation which is also considered in the analysis.

The graphs show that the peak memory allocation is stable and fairly low compared with the overall memory consumption; the graphs also show that the amount of garbage collection (indicated by the reduction in memory consumption) is clearly a function of the runtime of the application.

8.6.2.2 Impact of formula size

The influence of the formula size on memory allocation is shown in Figure 8.18. The numbers clearly show that verification is the only evaluation step that the formula size has an impact on, which coincides with the runtime behaviour as described in Section 8.6.1.2.

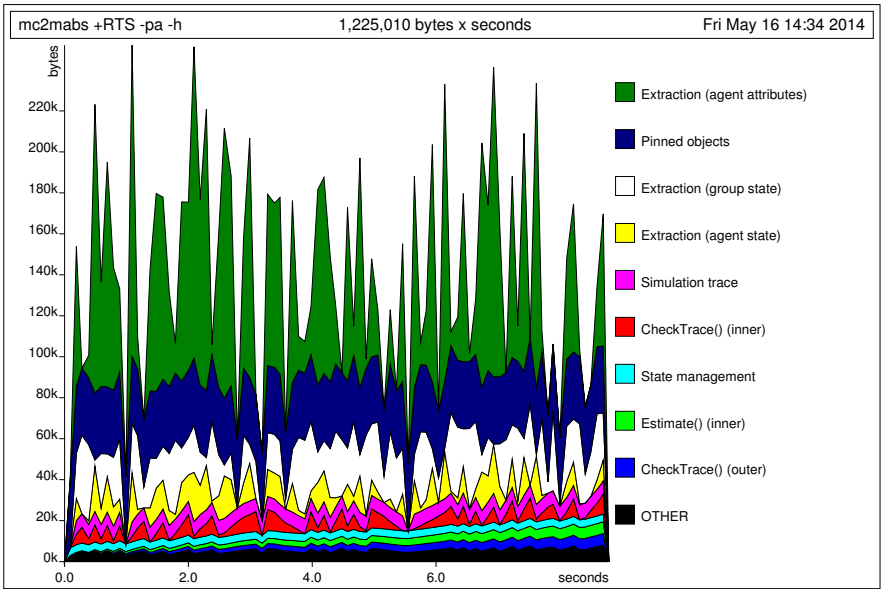


FIGURE 8.16: Runtime heap profile for 1,000 agents

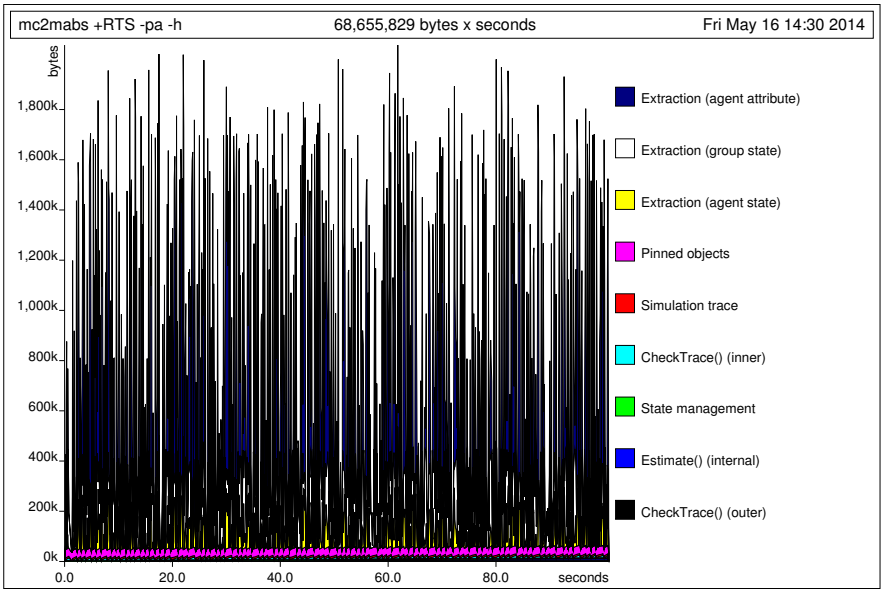


FIGURE 8.17: Runtime heap profile for 10,000 agents

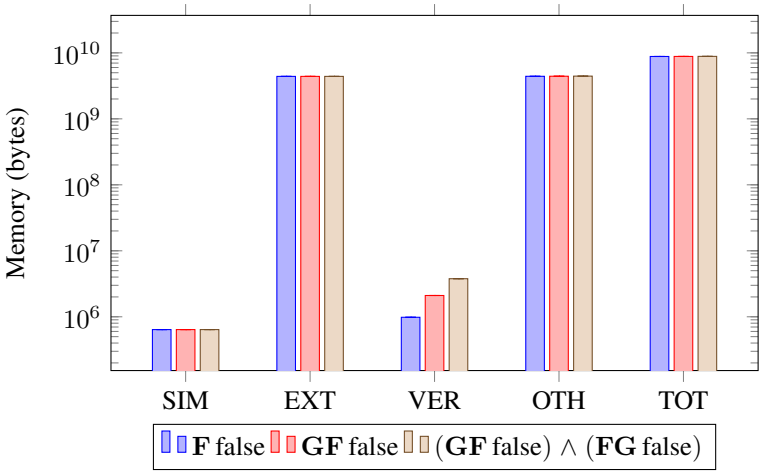


FIGURE 8.18: Influence of formula size on memory allocation for unquantified formulae

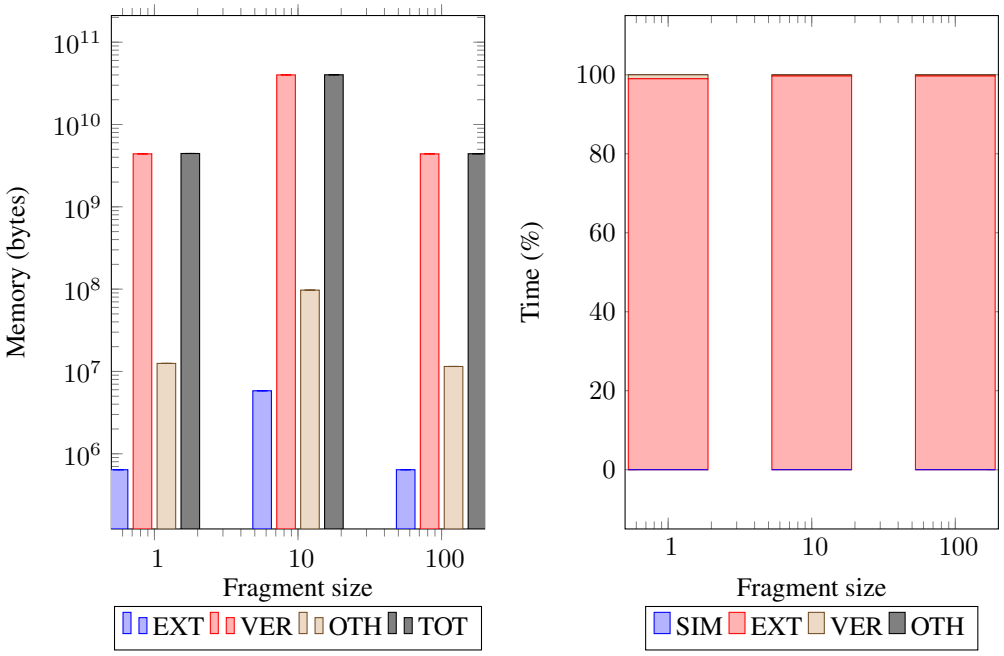


FIGURE 8.19: Influence of fragment size on absolute (left) and relative (right) memory allocation

8.6.2.3 Impact of fragment size

Finally, Figure 8.19 shows the impact of fragment size on memory allocation for formula '**G** true' and a population of 1,000 agents. The numbers show that fragment size has an impact on all evaluation steps. Similar to the runtime behaviour described in Section 8.6.1.3 (and for similar reasons), memory

consumption is highest in the case of fragment size 10. The chart on the right hand side also shows that, in the case of 1,000 agents, extraction is responsible for the vast majority of memory consumption; the fragment size has only a very small influence in this case.

8.7 Summary

This chapter gave an introduction of MC²MABS, a *practical tool* for statistical runtime verification of agent-based simulations. The tools consists of *two parts*: a *simulator* and a *monitor*. The simulator represents a framework which allows the user to formulate arbitrary model logic — either for the purpose of *simulation* or for the purpose of *replay* — in a compact way. The structure imposed by the framework reduces the need for the user to write model-independent boilerplate code and keeps the model description aligned with the semantic model required by the monitor. Models written in the simulator can thus be *analysed automatically*.

The monitor itself implements the algorithms described in Chapter 6 and is designed upon the principle of *lazy evaluation*. It triggers computations within the simulator if and only if they are *strictly necessary* for proving a given property, i.e. only if the property has not been either satisfied or refuted yet. Once a definite answer has been found, evaluation and therefore also simulation stops immediately.

Simulator and monitor communicate via a *functional interface*. It defines a set of *callback functions* that the user needs to implement in order to describe the model to be verified. They are called by the monitor whenever necessary.

The usage of MC²MABS was exemplified using a simple agent-based transmission model. The example involved the implementation of the model logic in the simulator as well as the formulation of a simple temporal property together with its evaluation. The description has been deliberately kept simple in order not to confuse the description of implementation aspects with the (potentially complex) logic of the underlying model.

A comprehensive evaluation of MC²MABS and its behaviour against the background of both runtime and memory consumption was given in Section 8.6.

The next chapter describes a more realistic and significantly more complex model from the agent-based modelling domain, along with a set of typical correctness properties.

Chapter 9

Case study

9.1 Introduction

This chapter describes the application of simLTL and the verification algorithms described in previous chapters to the automated evaluation of a real-world multiagent scenario using MC^{2}MABS. As opposed to the transmission model used as a running example in previous chapters, the model described in this chapter is of a different flavour and concerns a swarm of foraging robots. The model has been deliberately chosen to represent a technical rather than a social scenario in order to show the applicability of the described probabilistic analysis approach to a wider range of multiagent models.

The development of efficient realistic robot swarms is a common problem and has been discussed extensively in the literature [32, 28]. A typical characteristic of robot swarms is *self-organisation*, according to which the collective behaviour emerges from the behaviour of the individual robots. Emergent behaviour is generally seen to be irreducible to the constituent parts which makes the engineering of effective self-organising swarms a challenging task.

In this chapter, we touch upon the problem of robot swarms only very superficially. It is important to stress that we are not concerned with the development of an efficient robot swarm here; we use the robot swarm scenario as one of many typical instances of distributed problems which can be simulated and analysed efficiently as an agent-based model. Our main goal is to show the usefulness of statistical runtime verification during the development of agent-based models of different size and complexity.

To this end, we aim to illustrate how MC^{2}MABS can be used to formulate and verify different types of verification criterion rather than to provide a full analysis of a particular swarm robotic algorithm.

For the analysis of robot swarms, we can typically distinguish between three different types of approach: *experimentation* [146], *simulation* [160, 233] and *formal analysis* [159, 141]. In an *experiment*, real robots are placed within an environment and their behaviour is observed over time. It is obvious that, despite its realism, experimentation is often subject to hard time and resource constraints. A *simulation* can be seen as a computational surrogate for a real-world experiment which provides greater flexibility in terms of scale, complexity of the environment and exploration of possible scenarios. *Formal analysis* allows a modeller to obtain insights about the robot swarm analytically and can thus be seen as the most rigorous of the three approaches. It can be further subdivided into *microscopic* and *macroscopic* approaches. In a microscopic approach, individual robots are modelled explicitly, whereas macroscopic approaches focus on the aggregate behaviour of the entire swarm. What is, in most cases, common to both is that components are represented as stochastic processes which makes it possible to determine insights into the system's dynamics analytically.

The goal of this chapter is to show the usefulness of agent-based simulation in combination with statistical runtime verification for three different types of computational representation of robot swarms: a *macroscopic probabilistic model*, a *microscopic probabilistic model* and an *agent-based model*. The former two representations belong to the formal analysis domain; the latter follows the simulation approach and can be seen as a more sophisticated behavioural version of the microscopic model. As our underlying example, we use an adapted version of the robot swarm model presented by Liu *et al.* [159] which has also later been formally verified by Konur *et al.* [142]. We aim to show how properties about both macroscopic and microscopic models of robot swarms can be formulated in simLTL and verified efficiently using MC^{2}MABS.

The focus of the verification approach described in this thesis is on the analysis of *microscopic* models. Depending on their complexity, macroscopic models are often analytically tractable and thus amenable to formal mathematical analysis which, in terms of its rigour, reaches beyond the capabilities of a sampling-based approach. The analysis of macroscopic models described in Sections 9.3-9.3.2 thus serves a merely illustrative purpose and aims to show that statistical runtime verification can also be useful for those types of models, albeit to a lesser extent than purely formal techniques. The strength of the approach becomes more apparent in Section 9.4 and, in particular, in Section 9.5.

As described in Section 2.2.3, the focus of this work is on *internal validation*, i.e. the detection and assessment of *mechanisms* that are responsible for the temporal dynamics of a simulation model. Rather than with the link between the model and the real world (the purpose of external validation), the focus of internal validation is the link between the model and the underlying *theory*. Although the swarm models described in this chapter are clearly less theory-driven than, for example, many models in the area of social simulation, they still exhibit a range of mechanisms whose individual correctness and correct interplay may be usefully ascertained. Although this is less the case for the macroscopic models described in Section 9.3ff., the usefulness of internal validation becomes particularly apparent in the case of the individual-based models described in Sections 9.4 and 9.5.

Apart from the application of statistical model checking to the verification of different types of model, the chapter also illustrates a methodological aspect which can be usefully employed for the development of complex agent-based models. When building a complex, individual-based model, it can be useful to start with a simple probabilistic version in which most parameters are fixed and gradually replace them with more complex behaviours; verification can help in this process by quantifying the occurrence of behaviours and 'aligning' different versions of the same conceptual model. This process can also be reversed: transition probabilities detected through verification can be used to construct an abstract macroscopic model from a behavioural agent-based model which can then be analysed or — depending on its complexity — even formally verified in a non-approximate way.

The chapter is structured as follows. Section 9.2 introduces the problem domain and the model in an abstract, representation-independent way. Section 9.3 describes a macroscopic representation of the robot swarm and shows how typical aggregate correctness properties can be formulated and verified. Section 9.4 provides a simple microscopic representation of the model — a *microsimulation model* — in which robots are represented as simple, independent stochastic processes. We then show how MC^{2}MABS can be used to formulate and verify properties which cannot be verified in a purely macroscopic setting. Finally, Section 9.5 presents an agent-based representation of the robot swarm in which agents are situated in a real environment and interact indirectly by competing for randomly scattered food items; we show how MC^{2}MABS can be used to reveal some of the complex dynamics in the model and thus help to speed up the development process. The chapter concludes with a summary in Section 9.6.

All experiments were conducted on a Viglen Genie Desktop PC with four Intel® Core™ i5 CPUs (3.2 GHz each), 3.7 GB of memory and Gentoo Linux (kernel version 3.10.25) as operating system. Unless stated otherwise, the numbers are based on experiments involving 100 replications of the given model.

9.2 Problem description

Swarm robotics is a relatively new discipline which aims to harness ideas from biology for the engineering of complex multirobot systems. As opposed to centralised systems with pre-programmed complex behaviour, robot swarms consist of a large number of basic physical robots. Each individual robot follows a set of typically very simple, often threshold-based, rules. The swarm is able to *self-organise*, i.e. complex collective behaviour emerges from the actions and the interaction of the individual robots with each other and the environment. In many cases, individual behaviours are startlingly simple and follow, for example, flocking-based rules such as *cohesion*, *alignment* or *avoidance* [207].

Apart from their capability to exhibit emergent behaviour, robot swarms possess a number of characteristics which make them particularly interesting from an engineering point of view. For example, due to their distributed nature, swarms naturally exhibit a high level of redundancy and fault tolerance; given a sufficiently high number of robots, failure of an individual does not impact the overall behaviour of the swarm. Furthermore, swarms are inherently parallel and highly scalable; components can be added and removed without having to change the logic of the system. On the other hand, however, self-organisation — the most appealing feature of robot swarms — also represents one of its biggest problems. Due to the irreducibility of emergent behaviour to the behaviour of individual components, it is hard to engineer a swarm in such a way that a particular behaviour is guaranteed to be brought about; extensive analysis and verification is therefore critical.

In this chapter, we focus on *foraging*, a problem which has been widely discussed in the literature on *cooperative robotics* [45]. Foraging describes the process of a group of robots searching for food items, each of which delivers energy. Individual robots strive to minimise their energy consumption whilst searching in order to maximise the overall energy intake. The study of foraging is important because it represents a general metaphor to describe a broad range of (often critical) collaborative tasks such as *waste retrieval*, *harvesting* or *search-and-rescue*. A detailed overview of multirobot foraging is beyond the scope of this work; a good overview has been given by Cao *et al.* [45].

The model described in this chapter is based on the work of Liu *et al.* [159]. In the model, a certain number of food items are scattered across a two-dimensional space. Robots move through the space and search for food items. Once an item has been found, it is brought back to the nest and deposited which delivers a certain amount of energy to the robot. Each action that the robot performs also consumes a certain amount of energy. The model is deliberately kept simple. Each robot can be in one of five states: *searching* for food in the space, *grabbing* a food item that has been found, *homing* in order

```
void Agent::step(
  size_t confId,
  size_t tick,
  vector<Agent> const& population,
  boost::shared_ptr<Environment> const& env
)
{
  // calculate new values
  float _NS_0 = m_State[NR_4];
  float _NS_1 = (1.0-probF)*m_State[NS_0];
  float _NS_2 = (1.0-probF)*m_State[NS_1];
  float _NS_3 = (1.0-probF)*m_State[NS_2];
  float _NS_4 = (1.0-probF)*m_State[NS_3];

  float _NG_0 = probF*(m_State[NS_0]+m_State[NS_1]+m_State[NS_2]+m_State[NS_3]);
  float _NG_1 = (1.0-probG)*m_State[NG_0];
  float _NG_2 = (1.0-probG)*m_State[NG_1];
  float _NG_3 = (1.0-probG)*m_State[NG_2];
  float _NG_4 = (1.0-probG)*m_State[NG_3];

  float _ND_0 = probG*(m_State[NG_0]+m_State[NG_1]+m_State[NG_2]+m_State[NG_3]);
  float _ND_1 = m_State[ND_0];
  float _ND_2 = m_State[ND_1];
  float _ND_3 = m_State[ND_2];
  float _ND_4 = m_State[ND_3];

  float _NH_0 = m_State[NG_4] + m_State[NS_4];
  float _NH_1 = m_State[NH_0];
  float _NH_2 = m_State[NH_1];
  float _NH_3 = m_State[NH_2];
  float _NH_4 = m_State[NH_3];

  float _NR_0 = m_State[NH_4] + m_State[ND_4];
  float _NR_1 = m_State[NR_0];
  float _NR_2 = m_State[NR_1];
  float _NR_3 = m_State[NR_2];
  float _NR_4 = m_State[NR_3];

  // update state
  m_State[NS_0] = _NS_0;
  m_State[NS_1] = _NS_1;
  ...
}
```

LISTING 9.1: The agent logic implementation for the basic swarm model in MC^{2}MABS

to bring a food item back to the nest, *depositing* a food item in the nest, and *resting* in order to save energy. Transitions between states are probabilistic and either fixed or (which is clearly more realistic) dependent upon the state of the other agents, as described in the following section. The overall swarm energy is the sum of the individual energy levels.

9.3 Model 1: A difference equation approach

We start modelling the temporal dynamics of the robot swarm in a *macroscopic* way. Here, instead of representing each robot including its state transitions explicitly, the model describes the evolution of the *aggregate* behaviour of the system over time. If the resulting model is sufficient for the types of questions it is supposed to answer, then macroscopic modelling can represent a powerful solution to the state space explosion problem.

As mentioned above, the verification of a macroscopic model using MC^{2}MABS is only given for illustrative purposes. In cases where the model is tractable, formal, non-approximate verification such as exhaustive or symbolic model checking is clearly preferable over approximate verification. However, approximate verification of macroscopic models may, for example, be sensible if the model is too large to be analysed analytically. The purpose of this section is to show that MC^{2}MABS is not limited to the verification of purely agent-based simulation models and that it can also be used to answer certain properties about a macroscopic model.

The model used here has originally been presented by Liu *et al.* [159]; it has later been simplified and adapted for the purpose of macroscopic formal verification using PRISM by Konur *et al.* [142]. The authors present different models with increasing complexity. Starting with a basic model in which all transition probabilities are fixed, their values are subsequently replaced by functions over other variables in later versions the model. The model is further enhanced by additional variables which, for example, account for the density of robots in the environment.

For the purpose of illustration, we restrict our focus to the first two versions of the model. We start with the basic version in Section 9.3.1 which is then further extended in Section 9.3.2.

9.3.1 Scenario 1: Basic model with fixed transition probabilities

Model description

The macroscopic model is the result of a *counting abstraction* process; instead of modelling individual robots and their states over time, the model only keeps track of *how many* agents are in each possible state at any point in time. When representing the state space explicitly (as is the case in the original publication), the counting abstraction helps to transform a problem which is exponential in the number of agents into a polynomial one, which represents a significant advantage.

NS_i**:**

$$
\begin{aligned}
NS_0(t+1) &= NR_{T_{r-1}}(t) \\
NS_1(t+1) &= (1-\gamma_f)NS_0(t) \\
&\dots \\
NS_{T_{s-1}}(t+1) &= (1-\gamma_f)NS_{T_{s-2}}(t)
\end{aligned}
$$

NH_i**:**

$$
\begin{aligned}
NH_0(t+1) &= NG_{T_{g-1}} + NS_{T_{s-1}} \\
NH_1(t+1) &= NH_0(t) \\
&\dots \\
NH_{T_{s-1}}(t+1) &= NH_{T_{d-2}}(t)
\end{aligned}
$$

NG_i**:**

$$
\begin{aligned}
NG_0(t+1) &= \gamma_f \sum_{i=0}^{T_{s-2}} NS_i(t) \\
NG_1(t+1) &= (1-\gamma_g)NG_0(t) \\
&\dots \\
NG_{T_{s-1}}(t+1) &= (1-\gamma_g)NG_{T_{g-2}}(t)
\end{aligned}
$$

NR_i**:**

$$
\begin{aligned}
NR_0(t+1) &= NH_{T_{g-1}} + ND_{T_{s-1}} \\
NR_1(t+1) &= NR_0(t) \\
&\dots \\
NR_{T_{s-1}}(t+1) &= NR_{T_{d-2}}(t)
\end{aligned}
$$

ND_i**:**

$$
\begin{aligned}
ND_0(t+1) &= \gamma_g \sum_{i=0}^{T_{g-2}} NG_i(t) \\
ND_1(t+1) &= ND_0(t) \\
&\dots \\
ND_{T_{s-1}}(t+1) &= ND_{T_{d-2}}(t)
\end{aligned}
$$

FIGURE 9.1: Difference equations for the basic macroscopic robot foraging model [142]

The macroscopic model is defined by a set of variables $NS_i(t)$, $NG_i(t)$, $NH_i(t)$, $ND_i(t)$ and $NR_i(t)$ which describe the number of robots searching, grabbing, homing, depositing and resting, respectively, at time step t. Subscript i denotes the amount of time the robot has already spent in the respective state and ranges from 0 to T_s-1, T_g-1, T_h-1, T_d-1 or T_r-1, respectively. In the macroscopic model, effects resulting from competition between individual agents are abstracted away and represented as transition probabilities. For example, in the basic version of the model, each robot is assumed to have a certain probability γ_f of finding food which models the chance of perceiving food items in a real environment in an abstract way. Each robot also has a certain probability γ_g of grabbing food which corresponds with the idea of robots competing for mutually observed food items: multiple robots may detect the same food item but, depending on the amount of time it takes for a single robot to reach out for the item, only one will succeed.

Given these assumptions, the temporal dynamics of the model can then be described by a set of *difference equations* shown in Figure 9.1. The total number of agents being in a given state (irrespective of

how much time they already spent there) are calculated as follows:

$$N_s(t) = \sum_{i=0}^{T_{s-1}} NS_i(t) \qquad N_h(t) = \sum_{i=0}^{T_{h-1}} NH_i(t)$$
$$N_g(t) = \sum_{i=0}^{T_{g-1}} NG_i(t) \qquad N_r(t) = \sum_{i=0}^{T_{r-1}} NR_i(t)$$
$$N_d(t) = \sum_{i=0}^{T_{d-1}} ND_i(t)$$

Despite its equation-based nature, the model can be implemented in an agent-based way by assuming that the entire swarm is represented by a single agent with five states (searching, grabbing, homing, depositing and resting). The agent update logic is shown in Listing 9.1.

Each action (searching, grabbing, etc.) consumes a certain amount of energy which is denoted by variables E_s, E_g, E_h, E_r. Food items, on the other hand, are energy sources. Each time an agent deposits food, its energy level thus increases by an amount E_d which can be seen as the amount of energy gained minus the amount of energy consumed whilst depositing. The total energy of the swarm at time $t + 1$ is calculated as follows:

$$En(t+1) = En(t) + E_d ND_{T_{d-1}}(t) - E_s N_s(t) - E_g N_g(t) - E_r N_r(t) - E_h N_h(t) \tag{9.1}$$

Verification

Due to its lack of implemented mechanisms, the model has significantly weaker internal validation requirements than, for example, the individual-based versions described further below. Nevertheless, it is possible to ascertain some of the basic assumptions that form part of the theory behind the difference equations given above. First, it is naturally desirable for the swarm to gain more energy through food items than it loses during the foraging process. A designer of a robot swarm is thus interested in ensuring that, in the long run, the swarm energy will always be positive. This is a typical example of a *steady-state property* which can be answered with exhaustive (i.e. non-approximate) model checkers such as PRISM. Due to its focus on finite traces, a simulation-based model checker like `MC`^{2}`MABS` is not able to answer steady-state properties; it is, however, possible to get an *indication* of how the swarm energy level develops by stating (i) that it will never fall below zero, and (ii) that it will eventually always

```
int _aValue(int att, int id, int tick, Agent const& a)
{
  switch(att)
  {
    case SWARM_ENERGY:
    {
      double NS = a.m_State.find(NS_0)->second + a.m_State.find(NS_1)->second +
        a.m_State.find(NS_2)->second + a.m_State.find(NS_3)->second +
        a.m_State.find(NS_4)->second;
      double NG = a.m_State.find(NG_0)->second + a.m_State.find(NG_1)->second +
        a.m_State.find(NG_2)->second + a.m_State.find(NG_3)->second +
        a.m_State.find(NG_4)->second;
      double NR = a.m_State.find(NR_0)->second + a.m_State.find(NR_1)->second +
        a.m_State.find(NR_2)->second + a.m_State.find(NR_3)->second +
        a.m_State.find(NR_4)->second;
      double NH = a.m_State.find(NH_0)->second + a.m_State.find(NH_1)->second +
        a.m_State.find(NH_2)->second + a.m_State.find(NH_3)->second +
        a.m_State.find(NH_4)->second;
      energy += energyD*a.m_State.find(ND_4)->second - energyS*NS - energyG*NG -
        energyR*NR - energyH*NH;
      return floor(energy+0.5);
    }
    case NUM_AGENTS:
    {
      float res = 0;
      res += a.m_State.find(NS_0)->second;
      res += a.m_State.find(NS_1)->second;
      res += a.m_State.find(NS_2)->second;
      res += a.m_State.find(NS_3)->second;
      res += a.m_State.find(NS_4)->second;
      res += a.m_State.find(NR_0)->second;
      ...
      return floor(res+0.5);
    }
  }
}
```

LISTING 9.2: Implementation of the numeric functions for the basic swarm model

remain positive. In order to detect potential numerical instabilities, we may also want to ascertain that the total number of agents is always the same (e.g. 100). This leads to the formulation of the following three properties:

Property I :*"The total energy level will never fall below 0"*

Property II :*"The total energy will eventually always be positive"*

Property III :*"The total number of agents is always 100"*

Before describing the formulation of these properties in simLTL, we need to represent the variables mentioned (total energy level + total number of agents) as numeric functions in the simulation framework, which can be done by implementing function `_aValue` as described in Chapter 8. An example

implementation is shown in Listing 9.2. The simLTL formulae of both properties along with their machine-readable counterparts can now be given as follows:

$\mathbf{G}(SWARM\_ENERGY \geq 0)$ (Property I)

```
GALTL (AGlobally (AGEq (ANumFunc SWARM_ENERGY) (ANumVal 0)))
```

$\mathbf{FG}(SWARM\_ENERGY \geq 0)$[1] (Property II)

```
GALTL (AFinally (AGlobally (AGEq (ANumFunc SWARM_ENERGY) (ANumVal 0)))
```

$\mathbf{G}(NUM\_AGENTS = 100)$ (Property III)

```
GALTL (AGlobally (AEq (ANumFunc NUM_AGENTS) (ANumVal 100)))
```

Property	N	Total time	Result
I	100	0.003s	0.0
	200	0.003s	0.0
	300	0.003s	0.0
	400	0.003s	0.0
	500	0.003s	0.0
II	100	0.013s	1.0
	200	0.013s	1.0
	300	0.013s	1.0
	400	0.013s	1.0
	500	0.013s	1.0
III	100	0.010s	1.0
	200	0.010s	1.0
	300	0.010s	1.0
	400	0.010s	1.0
	500	0.010s	1.0

TABLE 9.1: Verification results for Properties I-III in the basic macroscopic model

Property	N	Initial swarm energy	Result
I	100	2,000	0.0
	100	2,500	0.0
	100	3,000	0.0
	100	3,500	0.0
	100	4,000	1.0

TABLE 9.2: Verification results for Property I and different values for the initial swarm energy

Before performing the verification, we need to think about the parametrisation of the model. There are various variables that may be varied during verification, for example γ_f, γ_g, E_0 (the initial energy level of the swarm) and N (the number of agents). For simplicity, we restrict our attention to the number of agents and choose $\gamma_f = 0.5$, $\gamma_g = 0.7$ and $N \in \{100, 200, ..., 500\}$. The results of the verification are shown in Table 9.1.

[1] As described in Section 5.2.2, particular care is advised when nested temporal properties are to evaluated upon finite traces since, in this case, their semantics differ significantly from their intuitive meaning.

Property I is never satisfied. This is due to the fact that, in the beginning of the simulation, the robots need some time to find, grab and deposit food before they gain their first amount of energy. Properties II and III are always satisfied which is reassuring. The verification results show two interesting technical effects. First, despite differences in swarm size, all experiments require an equal amount of time for simulation plus verification. This is due to the fact that, in the macroscopic case, 'simulation' reduces to simple numeric manipulation of agent cardinalities in different states and the number of actual agents in the swarm is negligible. Second, despite their apparent structural similarity, Property I can be answered much more efficiently than Property III. This is due to the fact that the energy level of the swarm will immediately fall below zero in the first tick. As opposed to Property III which requires examination of all group states, Property I can thus be immediately refuted.

The verification results show that the swarm does not have sufficient initial energy. Another set of experiments can help to clarify the initial energy level necessary in order to keep the swarm's overall energy level above zero. The results are shown in Table 9.2. they suggest that the swarm requires a total amount of approximately 4,000 ($\approx 3,950$ to be more precise) units of initial energy in order to satisfy Property I. In order to increase confidence in the results, we re-verified Property I based on 1,000 replications. Evaluation takes ≈ 16 seconds and returns a probability of 1.0. According to the sample size estimation procedure described in Section 6.4, we can now have 99% confidence that the probability of Property I is at least 96.5%.

Note how the properties given above clearly refer to the *model-theory link* rather than to the *model-phenomenon link*. We are not trying to *replicate* the dynamics of an existing swarm by means of simulation; instead, we are trying to come up with an efficient *design* of a swarm coordination mechanism that may, in future, be implemented physically. As a consequence, we are not interested in how well the model represents any real swarm (external validation); instead, we are interested in a confirmation of the theory as represented by the rules given above and its suitability for a future implementation (internal validation).

9.3.2 Scenario 2: Swarm model with variable transition probabilities

Model description

The second version of the swarm model replaces the fixed with variable transition probabilities. Instead of assuming a fixed probability of finding food, for example, this model assumes that the probability

```
void _preTick(size_t confId, size_t tick, vector<Agent> const& population)
{
  // determine number of foraging & grabbing agents
  vector<Agent>::const_iterator it = population.begin();
  float numForaging = it->m_State.find(NS_0)->second+it->m_State.find(NS_1)->second+
    it->m_State.find(NS_2)->second+
    it->m_State.find(NS_3)->second+it->m_State.find(NS_4)->second+
    it->m_State.find(NG_0)->second+it->m_State.find(NG_1)->second+
    it->m_State.find(NG_2)->second+it->m_State.find(NG_3)->second+
    it->m_State.find(NG_4)->second+it->m_State.find(ND_0)->second+
    it->m_State.find(ND_1)->second+it->m_State.find(ND_2)->second+
    it->m_State.find(ND_3)->second+it->m_State.find(ND_4)->second;

  float numGrabbing = it->m_State.find(NG_0)->second+it->m_State.find(NG_1)->second+
    it->m_State.find(NG_2)->second+it->m_State.find(NG_3)->second+
    it->m_State.find(NG_4)->second;

  // update transition probabilities
  probF = 1.0-( (alpha*numForaging)/100.0 );
  probG = 1.0-( (alpha*numGrabbing)/100.0 );
}
```

LISTING 9.3: Update of transition probabilities in function `_preTick`

is negatively proportional to the overall number of robots foraging. This corresponds with the intuitive idea that an increasing number of foraging robots decreases each individual robot's probability of finding food. The update rule for probability γ_f is given below:

$$\gamma_f(t) = \left(1 - \frac{\alpha N_{foraging}(t)}{N}\right) \tag{9.2}$$

where α is a *density factor*, $N_{foraging}(t) = N_s(t) + N_g(t) + N_d(t)$ denotes the number of foraging robots and N denotes the total number of agents.

The same principle can be applied to the grabbing probability. Instead of assuming a fixed probability, this version of the model assumes that the probability of grabbing food is a function of the total number of robots grabbing:

$$\gamma_g(t) = \left(1 - \frac{\alpha N_g(t)}{N}\right) \tag{9.3}$$

In order to adapt the transition probabilities accordingly in each time step in MC^{2}MABS, function `_preTick`, which is called once every tick immediately prior to the environment and population update, can be used. An example implementation is shown in Listing 9.3.

Verification

Now that transition probabilities are no longer fixed, the temporal dynamics of the robot swarm become harder to predict. Making transition probabilities dependent upon the number of agents being in a certain state which, in turn, itself depends upon the transition probabilities introduces a feedback loop which may influence the foraging efficiency of the entire swarm either positively or negatively. It is thus important for a designer to make sure that robots do not block each other.

There are clearly many different ways to ascertain this. We illustrate one particular example case by requiring that *"if the number of robots is more than n, then the average swarm energy exceeds E within t_A time steps"*. This can be formalised as follows:

$$\mathbf{G}(N \geq n \Rightarrow \mathbf{F}^{\leq t_A}(SWARM\_ENERGY \geq E)) \qquad \text{(Property IV}')$$

Values chosen for verification are $\alpha \in \{0, ..., 1\}$, $N \in \{20, 40, ..., 400\}$, $n = 100$, $E = 100 \times 10^3$ and $t_A = 100$. Due to its formulation over entire parameter *ranges*, this property requires a set of verification checks for different combinations of parameters (similar to the experiment verification performed for the basic model above). In PRISM, this can be achieved with the help of the internal experimental framework which checks the property above for all possible parameter combinations. The property is equally checkable in MC^{2}MABS with the help of *configurations*, although MC^{2}MABS does not currently provide a way of generating all possible configurations automatically; checking the property above would thus require the manual description of all parameter combinations as individual configurations. This is cumbersome but, depending on the number of combinations, still feasible in many cases. It is certainly possible to incorporate a more convenient mechanism but is outside the scope of this thesis.

It is important to note that simLTL does not currently provide support for bounded temporal operators as required by Property IV' above. However, since paths in MC^{2}MABS are by default time-bounded, the temporal check in the formula above can be solved in a purely semantic way by simply restricting the path length to t_A. Verifying the property above then amounts to verifying the following, slightly rewritten property upon each configuration in which $N \geq n$:

$$\mathbf{F}(SWARM\_ENERGY \geq E) \qquad \text{(Property IV)}$$

Property	N	Total time	Result
IV	100	0.010s	1.0
	200	0.007s	1.0
	300	0.007s	1.0
	400	0.003s	1.0
	500	0.003s	1.0
V	100	0.003s	1.0
	200	0.003s	1.0
	300	0.003s	1.0
	400	0.003s	1.0
	500	0.003s	1.0

TABLE 9.3: Verification results for properties IV (with $E = 10^5$) & V (with $t_A = 100$ and $n = 100$) in the macroscopic model with variable transition probabilities

Property	N	α	Result
IV	100	0.1	1.0
	100	0.2	1.0
	100	0.3	1.0
	100	0.4	0.0
	100	0.5	0.0

TABLE 9.4: Verification results for property IV and different values for the density factor α

The second additional property states that *"after t_A time steps, the number of foraging robots is always greater than m if the total swarm is larger than m"*. This aims to ensure that there is always a sufficient number of robots available for foraging:

$$\mathbf{G}(TICK \geq t_A \wedge N \geq n \Rightarrow NUM\_FORAGING \geq m) \qquad \text{(Property V')}$$

Similar to Property IV above, the property can be verified by examining the following, slightly rewritten version on all configurations in which the number of robots is greater than n:

$$\mathbf{G}(TICK \geq t_A \Rightarrow NUM\_FORAGING \geq n) \qquad \text{(Property V)}$$

For the verification experiments, we reset the initial swarm energy to 0 and choose the same parametrisation as in the previous example. The results are shown in Table 9.3. They show that Properties IV and V are always satisfied for swarm sizes between 100 and 500. It is obvious that the evaluation time of Property IV is dependent upon the size of the underlying population. This is due to the fact that, in the case of 100 agents, the desired energy level is reached significantly later than in the case of 500 agents. In other words, more ticks need to be simulated in the case of 100 agents than in the case of 500 agents.

We can also assess the impact of the density factor α on the overall development of the energy level. To this end, we can check Property IV with varying values for α, as shown in Table 9.4 for the case

of $N = 100$ and $\alpha \in \{0.1, 0.2, 0.3, 0.4\}$. The results suggest that increasing density of robots has a negative impact on the development of the overall energy level, a finding which is intuitively correct.

Let us summarise the experiments so far. The macroscopic model allows us to verify certain *aggregate* properties. We can show that the overall energy level of the swarm will not fall below 0 for a certain parametrisation and a given number of simulated ticks; we can also show that the total number of agents foraging will always be above a certain threshold during the simulation. What is impossible — or at least exceptionally hard — in most cases, however, is to provide deeper *explanations* for certain phenomena. For example, we cannot explain *why* the overall energy level falls below 0 as soon as the density increases, we can only hypothesise. It may be the case that, the more agents there are, the more likely they are to block each other which decreases their overall foraging capability and therefore also the total swarm energy. However, for a designer of a robot swarm, it is also important to be able to verify properties on a finer level of granularity. This requires an individual-based representation of the problem, a simple first version of which is described in the following section.

9.4 Model 2: A microsimulation approach

As explained in the introduction to this chapter, the macroscopic approach to modelling the foraging behaviour of a robot swarm as described in the previous section is largely motivated by the need to reduce the size of the resulting state space prior to formal verification. In general, a parallel composition of n independent components with s states each produces a state space that is in $O(s^n)$. In the case of 100 agents, each of which can be in one of 5 states, this amounts to $5^{100} \approx 7.89 \cdot 10^{70}$ states. In contrast, the macroscopic model based on counting abstraction described above comprises $O((n+t)^s)$ states where t is the number of timeout states. This represents a significant reduction in complexity, yet the approach still poses a number of problems. First, despite the change from exponential to polynomial complexity in the number of agents, the resulting model is still exponential in the number of states which renders its formal verification infeasible for non-trivial models. Second, counting abstraction only works if all agents within the model are entirely homogeneous; different individual behavioural protocols or transition probabilities render counting abstraction impossible. Third, as indicated in the final paragraph of the previous section, difference equations can become complicated to formulate as soon as the dynamics of the underlying model become nontrivial. Fourth, in a macroscopic model, all effects can only ever be dealt with in a 'mean-field way', i.e. by describing the average behaviour. This becomes particularly critical for the simulation of complex systems in which small local effects

(such as the probability of finding/grabbing food or collision avoidance described above) can amplify and have drastic consequences on the overall behaviour of the system. And finally, due to its global focus, a macroscopic model does not allow for the assessment of individual behaviour. Despite the overall energy level being positive, for example, there might still be individual agents for which the level reaches negative values. This is not detectable in a macroscopic model.

In this section, we translate the macroscopic model described before into a *microsimulation* representation and show how MC^{2}MABS can be used to verify properties which could not be verified before. As opposed to the macroscopic representation, a microsimulation model represents each agent individually and the overall behaviour of the system is determined by the union of the individual agents' behaviours. As opposed to a fully-fledged agent-based model (described in Section 9.5 below), on the other hand, a microsimulation does not model interactions between agents explicitly. Furthermore, the agents' behaviour is typically kept deliberately simple and represented by probabilistic state transitions, similar to the macroscopic case.

Despite their behavioural simplicity, however, microsimulations are able to overcome some of the limitations of purely macroscopic models and may exhibit significantly more complex temporal dynamics. Furthermore, due to their individual nature, they offer richer opportunities for probabilistic analysis, as shown in the following paragraphs. As opposed to the agent-based representation further below, the microsimulation model described here poses two major simplifications: (i) agent-to-agent interactions are not represented explicitly, and (ii) the model does not contain an explicit environment. Both simplifications are relaxed in Section 9.5. Due to its individual-based nature, the model offers significantly more possibilities for the interplay between individual agents and, as such, requires a higher level of internal validation than the macroscopic model described above. This is reflected in the higher number and higher complexity of correctness criteria formulable.

It is important to note that the model described here makes no claim to be exhaustive, not even to be overly realistic. Its mere purpose is to serve as a simple, intuitively comprehensible and fairly typical example of a microsimulation model. We are aware that, due to its simplicity (especially the homogeneous nature of agents and the lack of environmental influences), this very model could as well be analysed analytically; formal verification may thus seem slightly disproportionate. However, microsimulations do not require their agents to be homogeneous, nor do they exclude the modelling of environmental effects. Due to its reliance on mere simulation traces, on the other hand, MC^{2}MABS is entirely agnostic about the complexity of the underlying model and is equally capable of analysing a significantly more

complex microsimulation model. In order to focus on the verification, however, the model described here has been deliberately kept simple.

The purpose of this section is two-fold: first, we aim to demonstrate that the same set of verification criteria used for the macroscopic model can also be used to verify the dynamics of the microsimulation model and show their behavioural similarity. In this case, verification can be seen as ensuring that both models are *aligned* properly. And second, we aim to demonstrate the usefulness of `MC`^{2}`MABS` for the verification of *individual* properties which cannot be verified on a macroscopic model.

For simplicity, we restrict the description to the basic model, i.e. to the one with fixed food finding and grabbing probability.

Model description

We start the description by replacing the single 'population agent' from the previous section with a set of individual agents, each of which follows the simple probabilistic protocol given below.

- If the agent is $searching$, then start $grabbing$ with probability γ_f and remain $searching$ with probability $1 - \gamma_f$. If no food can be found within T_s time steps, then start $homing$.
- If the agent is $grabbing$, then start $depositing$ with probability γ_f and remain $grabbing$ with probability $1 - \gamma_f$. If no food can be grabbed within T_g time steps, then start $homing$.
- If the agent is $depositing$, then start $resting$ after T_d time steps.
- If the agent is $homing$, then start $resting$ after T_h time steps.
- If the agent is $resting$, then start $searching$ after T_r time steps.

In order to implement the model, the agent needs two state variables: one to represent its 'working state' (searching, grabbing, etc.) and one to represent the time it has already spent in the current working state. We denote the former with $STATE$, the latter with $COUNT$. We also assume that each agent holds its own energy level which is adjusted as it moves through its state space. We denote this state variable with $ENERGY$. The behavioural protocol can then be implemented as shown in Listing 9.4.

```
void Agent::step(
  size_t confId,
  size_t tick,
  vector<Agent> const& population,
  boost::shared_ptr<Environment> const& env
)
{
  if(m_State[STATE] == SEARCHING) {
    m_State[ENERGY] -= energyS;
    if(++m_State[COUNT] == timeS) {
      m_State[COUNT] = 0;
      m_State[STATE] = HOMING;
    }
    else {
      if(getRand() <= probF) {
        m_State[COUNT] = 0;
        m_State[STATE] = GRABBING;
      }
    }
  }
  else if(m_State[STATE] == GRABBING) {
      m_State[ENERGY] -= energyG;
      if(++m_State[COUNT] == timeG) {
        m_State[COUNT] = 0;
        m_State[STATE] = HOMING;
      }
      else {
        if(getRand() <= probG) {
          m_State[COUNT] = 0;
          m_State[STATE] = DEPOSITING;
        }
      }
  }
  else if(m_State[STATE] == HOMING) {
      m_State[ENERGY] -= energyH;
      if(++m_State[COUNT] == timeH) {
        m_State[COUNT] = 0;
        m_State[STATE] = RESTING;
      }
  }
  else if(m_State[STATE] == DEPOSITING) {
      if(++m_State[COUNT] == timeD) {
        m_State[ENERGY] += energyD;
        m_State[COUNT] = 0;
        m_State[STATE] = RESTING;
      }
  }
  else if(m_State[STATE] == RESTING) {
      m_State[ENERGY] -= energyR;
      if(++m_State[COUNT] == timeR) {
        m_State[COUNT] = 0;
        m_State[STATE] = SEARCHING;
      }
  }
}
```

LISTING 9.4: Implementation of the individual agents' behavioural protocol for the basic microsimulation model

Verification

The microsimulation model allows us to verify the same properties as described in the previous section, albeit in a slightly different way. Consider again the property that we used in the macroscopic model in order to verify that the swarm energy will always be positive:

$$\mathbf{G}(SWARM\_ENERGY \geq 0) \qquad \text{(Property I)}$$

```
GALTL (AGlobally (AGEq (ANumFunc SWARM_ENERGY) (ANumVal 0)))
```

In the machine-readable version of simLTL, it becomes apparent that the formula is an agent formula masked as a full simLTL formula. This is necessary since the macroscopic model only consists of a single agent. As a consequence, $SWARM\_ENERGY$ is implemented as a numeric function on the agent level. The swarm energy itself was represented as a function of the number of agents being in certain states and the energy consumed within those respective states (see Equation 9.1).

In the microsimulation case, we now have a set of individual agents which are represented explicitly; the total swarm energy level can therefore be calculated by simply adding up the individual agent levels:

$$En = \sum_{i=1}^{N} ENERGY_i \tag{9.4}$$

The total swarm energy can be seen as an attribute of the entire population and is therefore implemented on the group level using function `_gValue`, as shown in Listing 9.5. Property I from above can now be adapted as follows for the microsimulation case (we only give the machine readable version here — the mathematical notation is equivalent to the one from the macroscopic model):

```
GGlobally (GGEq (GNumFunc SWARM_ENERGY) (GNumVal 0))
```

Properties II & III can be adapted accordingly. Before verifying Property I and focussing more on the energy level, however, we want to make sure that the implemented model is internally correct. In addition to global properties, the microsimulation model allows us to formulate more fine-grained properties about individual agents and ascertain their correctness. It may, for example, be useful to ensure that the individual agents comply with the probabilistic protocol shown above. This can be done

```
int _gValue(int att, int tick, vector<Agent>& population) {
switch(att) {
  case SWARM_ENERGY:
  {
    energyN = 0;
    int cnt=0;
    for(vector<Agent>::iterator it=population.begin(); it!=population.end(); ++it) {
      energyN += it->m_State[ENERGY];
      cnt++;
    }
    return energyN;
  }
  case NUM_AGENTS:
    return population.size();
  }
  return 0;
}
```

LISTING 9.5: Implementation of function `_gValue` for the basic version of the microsimulation model

by proving that their expected probability of transitioning, for example, from searching to grabbing coincides with the valuation of parameter γ_f. The usefulness of this check may not be entirely obvious in this version of the model, since the probabilities are hardcoded. However, in subsequent versions where transition probabilities themselves depend upon other values, determining their values can be of great help to the modeller (this is demonstrated further below and in Section 9.5). But even in the basic model described here, numeric inconsistencies (e.g. rounding or casting errors between integer and floating point numbers) may cause the resulting probability to deviate from what was originally intended. Ensuring the correctness of transition probabilities through appropriate verification can thus help to increase the confidence in the simulation outcome.

As described in Section 7.5.2, transition probabilities can be verified using `MC`^{2}`MABS` by means of *quantitative conditional reasoning* on trace fragments of size 2. Following the description in that section, quantitative conditional reasoning requires the comparison of different probabilities, each of which results from a separate configuration. We illustrate the formulation for the transition probability from searching to grabbing. Let us first denote the property of interest which is given in natural language below with ψ_{SG}.

$$\psi_{SG} = \textit{“An agent's expected probability of transitioning from searching to grabbing is 0.5”}$$

In order to verify this property, we need two subformulae which are given below[2].

[2]Remember that, since the formula is never being evaluated upon the final state of a trace, both the weak and the strong version of the ‘next’ operator can be chosen. This is denoted by the unquantified operator in the formula.

```
const char* getFormula(int idx)
{
  stringstream ss;
  string f;

  switch(idx) {
    case 1:
      return "GALTL (AAnd (AEq (AAttribute STATE) (ANumVal SEARCHING) )
        (AOr ALast (ANext (AEq (AAttribute STATE) (ANumVal GRABBING)))))";
    case 2:
      return "GALTL (AEq (AAttribute STATE) (ANumVal SEARCHING) )";
  }
}
```

LISTING 9.6: Implementation of the two configuration formulae for verifying the correctness of the state transition from searching to grabbing

$$\psi_1 = (STATE = SEARCHING) \wedge \mathbf{X}(STATE = GRABBING)$$

$$\psi_2 = (STATE = SEARCHING)$$

The overall transition probability is then calculated as follows[3]:

$$Pr_2(\psi_{SG}) = Pr_2(\psi_1)/Pr_2(\psi_2) \qquad \text{(Property VI)}$$

Those calculations can be easily integrated in MC^{2}MABS. Listing 9.6 shows the implementation of the two simLTL formulae ψ_1 and ψ_2 and their association with configurations; the formulation of the final property ψ which utilises the two formulae and combines them into one overall statement corresponding with Property VI is shown in Listing 9.7.

The verification results for Property VI are shown in Table 9.5. Evaluation was performed by checking ψ_{SG} on 100 replications of the underlying simulation. The results strongly indicate that the expected transition probability of individual agents exhibited by the model at runtime complies with the desired value. The correctness of the model with respect to γ_g — the probability of moving from grabbing to depositing, denoted $Pr_2(\psi_{GD})$ — can be verified accordingly; the results are shown at the bottom of Table 9.5[4].

[3] As described in Section 3.6, we denote with the subscript number the fragment size. We further use t to denote the evaluation upon entire traces.

[4] The verification time for Properties ψ_1 and ψ_1 are not available separately because they are subsumed in the overall verification time of Property ψ_{SH}.

```
const char* getProperty()
{
  string conf4 = getConf(
    1,
    getCheckingMode(),
    getFragmentSize(),
    getNumAgents(),
    getNumAgentAtts(),
    getNumTicks(),
    getNumReps()
  );

  string conf5 = getConf(
    2,
    getCheckingMode(),
    getFragmentSize(),
    getNumAgents(),
    getNumAgentAtts(),
    getNumTicks(),
    getNumReps()
  );

  stringstream ss;
  ss << "Div (" << conf4 << ") (" << conf5 << ")";

  return ss.str().c_str();
}
```

LISTING 9.7: Implementation of the final state transition property

Whereas the previous checks helped to *confirm* the correctness of transition probabilities that were known before, the same idea can be used to *detect* transition probabilities that were not built into the model explicitly and are thus harder to obtain analytically. For example, we can estimate the probability of an individual timeout by estimating and summing up the probabilities of an individual agent's transitioning from searching to homing and from grabbing to homing. This, in turn, can give an indication of potential design flaws. Let therefore $Pr_2(\psi_{SH})$ denote the individual probability of moving from searching to homing and $Pr_2(\psi_{GH})$ the individual probability of moving from grabbing to homing. Both probabilities are calculated similarly to $Pr_2(\psi_{SG})$ above. The probability of a timeout, denoted $Pr_2(\psi_{timeout})$, is then calculated as follows:

$$Pr_2(\psi_{timeout}) = Pr_2(\psi_{SH}) + Pr_2(\psi_{GH}) \qquad \text{(Property VII)}$$

The results are shown in Table 9.6[5]. They indicate that the probability of reaching a timeout is $\approx 3.9\%$, the criticality of which can then be judged by the modeller.

[5]The verification time for Property $\psi_{timeout}$ is not available because its probability has been determined by manually calculating $Pr_2(\psi_{SH}) + Pr_2(\psi_{GH})$.

Property	N	Total time	Result
ψ_1	100	n/a	0.7055
ψ_2	100	n/a	0.14396
ψ_{SG}	100	39.52s	0.49006
ψ_{GD}	100	43.08s	0.67941

TABLE 9.5: Verification results for Property VI (or ψ_{SG}) and ψ_{GD}

Property	N	Total time	Result
ψ_{SH}	100	39.043s	0.03312
ψ_{GH}	100	38.614s	0.0061
$\psi_{Timeout}$	100	n/a	0.0392

TABLE 9.6: Verification result for Property VII

As indicated above, given the simplicity of the model used here, the same insights could have also been derived analytically. Remember, however, that microsimulation models are generally likely to be significantly more complex and contain heterogeneous agents as well as environmental stimuli.

After ascertaining the correctness of individual state transitions, we can now turn our attention to the foraging behaviour of individual agents. In Section 9.3.1 above, we showed that the macroscopic model requires slightly less than 4,000 units of initial energy in order to ensure that the overall energy level of the swarm never falls below zero. The problem with the macroscopic model is that it only considers the *average* behaviour of the system; the variance that a real system would exhibit remains undetected. We should therefore rephrase the statement above as follows: the model requires slightly less than 4,000 units of initial energy in order to ensure that the *average* overall energy level of the swarm never falls below zero. In cases where a swarm running out of energy represents a serious technical problem, pure average-case analysis may not be sufficient.

The microsimulation can help us to perform deeper analyses. Let us therefore re-verify Property I, $\mathbf{G}(SWARM\_ENERGY \geq 0)$, on the microsimulation model, after the initial energy level has been adjusted accordingly. The microsimulation model is an individual-based model, we therefore need to think about how the initial energy should be distributed. In the following experiment, the initial energy of 3,950 units is distributed equally to all agents; each agent thus comprises an initial energy level of 39.5 units. In addition to the results of the verification, we can also use the simulation framework to determine the variance of the energy level. All we need to do is to determine the minimum, the average and the maximum energy level across all runs by tracking its evolution over time (i.e. every time the energy value is calculated in function `_gValue`) and output the values in the end of the simulation (e.g. in function `_postConf`. The results of a verification experiment based on a population of 100 agents, 1,000 ticks and 1,000 replications are shown below[6]. The experiment took 1 minute and 17 seconds to complete (including simulation).

[6]We write Pr_t in order to denote the evaluation of the property on full traces.

Min. energy: -298

Avg. energy: 26,402

Max. energy: 60,012

$Pr_t(\mathbf{G}(SWARM\_ENERGY \geq 0))$**:** 0.528

The results indicate that the overall energy level only remains positive in $\approx 50\%$ of all runs. This supports the verification results of the macroscopic model, yet it also shows that, in 50% of all cases, the swarm will run out of energy (albeit only slightly), as the minimum energy value indicates. We can run the experiment again by giving each robot slightly more initial energy (45 units). The results are shown below.

Min. energy: 72

Avg. energy: 50,162

Max. energy: 59,318

$Pr_t(\mathbf{G}(SWARM\_ENERGY \geq 0))$**:** 1.0

By giving each individual agent 45 units of initial energy, we can now have more confidence that the overall swarm energy will never fall below zero (during the course of the simulation!). It is interesting to note that, in this case, simulation plus verification of Property I took 2 minutes and 32 seconds to complete (instead of 1 minute and 17 seconds before). This is due to the fact that, as opposed to the previous check, the formula cannot be refuted before the end of the simulation.

Despite the energy level of the entire swarm constantly being above zero, however, there may well be individual agents for which the energy level occasionally falls below 0. We can test this by stipulating that *"for all agents, the energy level will never fall below zero"*. This can be formulated in simLTL as follows:

$\forall(\mathbf{G}(ENERGY \geq 0))$ (Property VIII)

```
GForall (GALTL (AGlobally (AGEq (AAttribute ENERGY) (ANumVal 0))))
```

Verification of this property takes approximately 2.9 seconds and returns a probability of 0 which suggests that there are agents with negative values in every single run. Verification is very quick which

suggests that the formula could be refuted early in the process. We may now be interested in the *expected probability* of a single agent reaching a negative energy level. As described in Section 7.3, expected probabilities of individual agents can be obtained by verifying unquantified agent formulae on randomly selected agent traces. Since all formulae need to be full simLTL formulae, however, the desired agent formula needs to be wrapped into `GALTL` as shown below.

$$\mathbf{G}(ENERGY \geq 0) \qquad \text{(Property IX)}$$

```
GALTL (AGlobally (AGEq (AAttribute ENERGY) (ANumVal 0)))
```

Verification of this property takes 57 seconds and returns a probability of 0.31. Remember that the property stipulates that an agent's energy level will *always stay above 0*. In order to determine its probability of reaching a negative energy level, we thus need to take the inverse probability, i.e. $1.0 - 0.31 = 0.69$. We can thus conclude that, in the current version of the swarm model, an individual robot has a probability of almost 70% to run out of energy.

Let us summarise the results so far. The macroscopic model described in Section 9.3 allowed us to verify properties about the mean-field behaviour of the robot swarm. Through verification, we could, for example, detect that the overall energy level of the swarm can only be kept positive if an initial energy of approximately 4,000 units is provided. This helped us to establish a causal relationship between the initial amount of energy and its behaviour over the simulation period. Verification of the microsimulation model described in this section allowed us to go further and verify additional, more fine-grained properties. We could show that, in 50% of all cases, the overall energy level of the swarm will eventually fall below 0. We could further show that, even in cases where the swarm has a 100% chance of always having enough energy, there may be a significant portion of individual agents that will eventually run out of energy. In that way, we are not only able to understand causal relationships between the initial values and the swarm as a whole, but also between the initial values and individual agents.

In order to reach a level where all agents have enough energy throughout their lifetime, additional analyses are necessary. They are described in the next section where we further increase the complexity of the model and make it properly agent-based.

9.5 Model 3: An agent-based approach

The previous section described a microsimulation representation of the robot swarm scenario. As opposed to the macroscopic difference equations approach, the microsimulation model represents each agent individually. What distinguishes the microsimulation model from a fully-fledged agent-based model, is its lack of interaction between agents and its focus on purely probabilistic individual decision making as opposed to a more behavioural, rule-driven approach. The purpose of this section is to address these points and transform the microsimulation representation of the swarm scenario into an agent-based one. We then show how $\texttt{MC}^2\texttt{MABS}$ can be used during the development process to verify properties about the model, to obtain transition probabilities and state distributions and to understand better its mechanisms for the purpose of internal validation.

Model description

Instead of viewing a population of robots as an abstract entity in which agents have a certain probability of finding food, the agent-based approach aims to model the world which is inhabited by the robots explicitly. It is common to many agent-based models to model the environment as a two-dimensional 'grid world'. In this model, we define the world that the agents inhabit as a grid of 100×100 cells. Each grid cell can be inhabited by an arbitrary number of agents. Food items are distributed uniformly across the grid. Again, similar to the microsimulation case, our goal is not to construct an overly realistic model here. Since the focus is on illustration, the model is thus kept deliberately simple.

The agents themselves still largely follow the behavioural protocol from the previous section, yet with a number of slight modifications. Since agents are now living in a spatial environment, they need the ability to move through the grid. Apart from their 'working state' (searching, grabbing, etc.), their energy level and the counter which determines how much time an agent spends in a certain state, we add two state variables $POSX$ and $POSY$ which describe the cell that the agent currently inhabits. Furthermore, in order to make things more interesting, we assume that there are initially two different types or *makes* of agent which only differ in terms of their field of vision. In order to reflect this, we also add a variable $MAKE$ to the agent's internal state.

As mentioned above, food items are distributed uniformly across the grid. In the current version of the model, there are 1,000 food items distributed across 10,000 grid cells, which amounts to a food density of 10%. Agents of make 0 are able to detect all food items within a radius of 1, agents of make 1 are

able to detect all food items within a radius of 4. The behavioural protocol that each agent follows is shown below.

- If the agent is $searching$, then look for food. If food has been found, move to the cell and start $grabbing$; otherwise remain $searching$. If no food can be found within T_s time steps, then start $homing$.
- If the agent is $grabbing$ and, after T_g time steps, the food is still there, then grab it and start $depositing$; otherwise start $homing$.
- If the agent is $depositing$, then start $resting$ after T_d time steps.
- If the agent is $homing$, then start $resting$ after T_h time steps.
- If the agent is $resting$, then start $searching$ after T_r time steps.

The agent-based version of the swarm model makes two of the previously explicitly represented aspects implicit, i.e. emerging from the implemented behavioural rules. First, the probability of finding food is now based on a robot's *vision*: an agent starts grabbing if and only if a food item has been found in its direct neighbourhood. And second, the probability of grabbing food is now a result of *competition*: when an agent finds food, it attempts to grab it; if, after T_g time steps have passed, another agent has grabbed the food item in the meantime, the first agent misses out and starts homing.

The implementation of the agents' behavioural protocol is shown in Listing 9.8. Function `findFood/5`, which is not further described here, finds a random food item within the agent's current field of vision. The function returns true if an item has been found, false otherwise. The position of the food item is returned via the last two function arguments.

Verification

The agent-based model differs significantly from the previous ones in that agents are now inhabiting a spatial environment to which they react. Due to situatedness and implicit competition for food, the dynamics of the model become harder to predict. The first significant difference from both the macroscopic and the microsimulation model is that agents no longer have fixed transition probabilities which they follow; the probability of transitioning from one state into another is now dependent upon the situation that each robot finds itself in. As a consequence, the model now contains real *causal mechanisms*

```
void Agent::step(
 size_t confId,
 size_t tick,
 vector<Agent> const& population,
 boost::shared_ptr<Environment> const& env
)
{
 if(m_State[STATE] == SEARCHING) {
  m_State[ENERGY] -= energyS;
  if(++m_State[COUNT] == timeS) {
    m_State[COUNT] = 0;
   m_State[STATE] = HOMING;
  }
  else {
   float posX = m_State[POSX], posY = m_State[POSY];
   bool foodFound = false;
   float foodX, foodY;
   // if food found => grabbing
   if(findFood(posX, posY, m_State[MAKE], foodX, foodY)) {
    m_State[POSX] = foodX;
    m_State[POSY] = foodY;
    m_State[COUNT] = 0;
    m_State[STATE] = GRABBING;
    foundFood[m_State[MAKE]]++;
   }
   else {
    // move randomly one step
    size_t stepX = floor(getRand()*100+0.5);
    size_t stepY = floor(getRand()*100+0.5);
    stepX = (stepX%3)-1;
    stepY = (stepY%3)-1;
    m_State[POSX] = ((size_t)m_State[POSX]+stepX)%100;
    m_State[POSY] = ((size_t)m_State[POSY]+stepY)%100;
   }
  }
 }
 else if(m_State[STATE] == GRABBING) {
   m_State[ENERGY] -= energyG;
   if(++m_State[COUNT] == timeG) {
    size_t posX = m_State[POSX];
    size_t posY = m_State[POSY];
    // if food is still there, grab it
    if(food[posX][posY] > 0) {
     food[posX][posY]--;
     m_State[COUNT] = 0;
     m_State[STATE] = DEPOSITING;
    }
    // otherwise move to homing
    else {
     m_State[COUNT] = 0;
     m_State[STATE] = HOMING;
    }
   }
 }
 else if(m_State[STATE] == HOMING) {
   m_State[ENERGY] -= energyH;
   if(++m_State[COUNT] == timeH) {
    m_State[COUNT] = 0;
    m_State[STATE] = RESTING;
   }
 }
 else if(m_State[STATE] == DEPOSITING) {
   if(++m_State[COUNT] == timeD) {
    m_State[ENERGY] += energyD;
    m_State[COUNT] = 0;
    m_State[STATE] = RESTING;
   }
 }
 else if(m_State[STATE] == RESTING) {
   m_State[ENERGY] -= energyR;
   if(++m_State[COUNT] == timeR) {
    m_State[COUNT] = 0;
    m_State[STATE] = SEARCHING;
   }
 }
}
```

LISTING 9.8: Implementation of the individual agents' behavioural protocol for the basic agent-based model

Transition	Total time	Result
$S \rightarrow G$	1m 52s	0.1701
$G \rightarrow D$	2m 01s	0.1735
$G \rightarrow H$	2m 09s	0.0076
$D \rightarrow R$	2m 02s	0.1868
$R \rightarrow S$	2m 03s	0.1913

TABLE 9.7: Transition probabilities for all agents in the agent-based model

Make	Transition	Total time	Result
0	$S \rightarrow G$	2m 40s	0.0363
	$G \rightarrow D$	2m 40s	0.1945
	$G \rightarrow H$	2m 46s	0.0034
	$D \rightarrow R$	2m 50s	0.1912
	$R \rightarrow S$	2m 55s	0.1950
1	$S \rightarrow G$	2m 50s	0.6223
	$G \rightarrow D$	2m 40s	0.1678
	$G \rightarrow H$	2m 42s	0.0090
	$D \rightarrow R$	2m 42s	0.2111
	$R \rightarrow S$	2m 47s	0.1833

TABLE 9.8: Separate transition probabilities for agents of make 0 and 1

whose correctness needs to be ascertained in order to deem the model internally valid. The presence of explicit mechanisms opens up new possibilities for correctness assessment and thus increases the extent of internal validation significantly.

As in the previous sections, our goal is to determine whether the model is reasonably robust, i.e. whether agents have enough energy during the simulated timespan. Based on the findings of the previous models, the initial version of the agent-based model has the following parametrisation:

- 100 agents, 1,000 ticks
- Time spent in each state: $T_s = T_g = T_r = T_h = T_d = 5$
- Energy consumed in each state: $E_s = 12, E_g = 12, E_h = 6, E_r = 2, E_d = 62$
- Initial level of energy per agent: 40

In order to check whether this parametrisation already satisfies the requirements, we first verify Property I on the agent-based model which states that the swarm as a whole will never run out of energy. Despite every robot having 40 units of initial energy, however, the verification returns a probability of 0 which shows that the parametrisation of the microsimulation is not suitable for this version of the model.

In order to gain a deeper understanding of why this may be the case, it is useful to study how frequently robots switch from one state into another by determining their expected transition probabilities. This can be done similarly to the microsimulation case. The results for 100 replications are shown in Table 9.7[7]. We can see that robots have an equal probability of finding and grabbing food ($\approx 17\%$). We can

[7]For clarity, we abbreviate states with their capitalised first letters in all subsequent tables.

Vision	$Pr_2(\psi_{SG})$	$Pr_2(\psi_{GD})$	$Pr_t(\mathbf{G}(SWARM\_ENERGY \geq 0))$	$Pr_2(D \wedge \mathbf{X}R)$
1	0.3416	0.1889	0.0	0.0108
2	0.1344	0.1853	0.0	0.0300
3	0.3735	0.1711	0.0	0.0438
4	0.6571	0.1631	0.0	0.0460
5	0.8364	0.1630	0.0	0.0470

TABLE 9.9: Expected individual transition probabilities and probability of constant positive swarm energy

also see that agents have a very low probability of transitioning into the homing state, which is positive since homing is always caused by a timeout and thus undesirable.

When calculating the transition probabilities, however, we need to take into account that we now have two different makes of agent, each of which can be expected to have different probabilities. In order to get a more detailed view, we thus need to 'zoom in' and re-calculate the transition probabilities separately for each group. This can be achieved by using the selection operator $\langle\langle\rangle\rangle$ (see Section 5.4.2). For robots of make 0, for example, the properties necessary for calculating the transition probability from searching to grabbing can be formulated as follows:

$$\psi_1 = \langle\langle MAKE == 0 \rangle\rangle [(STATE = SEARCHING) \wedge \mathbf{X}(STATE = GRABBING)]$$
$$\psi_2 = \langle\langle MAKE == 0 \rangle\rangle [(STATE = SEARCHING)]$$

Similar to the microsimulation case, the overall transition probability can then be calculated as described for Property VI in Section 9.4. The results for all checks are shown in Table 9.8. It is obvious that robots of make 1 have a significantly higher probability of finding food which, given their significantly larger field of vision, is intuitively correct. What is also interesting, however, is that a robot's make seems to have a small but obvious impact on its probability of grabbing food; this is indicated by the lower probability of transitioning from grabbing to depositing for robots of make 1. One possible explanation is that, due to their larger field of vision and their consequently higher probability of finding food, robots of make 1 may block each other by 'stealing' food that is already aimed for by a different robot. This explanation may also be underpinned by the slightly higher probability of robots of make 1 moving from grabbing to homing than robots of make 0: the only reason for performing this transition is that a food item aimed for is lost to a different agent. Given the small sample size, however, care needs to be taken when interpreting the numbers — especially when differences are very small, as in this case.

The numbers seem to suggest that the size of the field of vision has a positive impact on the food finding probability and a slightly negative impact on the food grabbing probability. This hypothesis can be investigated further by performing a range of experiments in which the vision parameter is constantly increased. The results are shown in Table 9.9. The numbers in the second column indicate that, in fact, the size of the field of vision has a significant positive impact on the probability of finding food (as expected). As described in Section 7.6, this proves that there is a *causal dependence* between an agent's field of vision and its probability of finding food. The numbers in the third column indicate that there is a slightly negative correlation between the size of the field of vision and the probability of grabbing food. The fourth column of the table shows the probability of Property I which is 0 in all cases; varying the field of vision alone is thus not sufficient for sustaining a positive energy level (at least not in the current scenario).

The numbers suggest that, despite the slight loss in grabbing probability, the swarm designer is best off by giving all robots a high field of vision. In order to confirm this assumption, we can formulate another property which denotes the *overall probability of an agent gaining energy*. Remember that energy is always gained in the final time step of the depositing state, i.e. before the agent starts resting. In order to determine the overall probability of an agent gaining energy, we can thus formulate the following property:

$$(STATE = DEPOSITING) \wedge \mathbf{X}(STATE = RESTING)^{8} \qquad \text{(Property X)}$$

Since this is a property whose truth needs to be ascertained on state transitions, it needs to be checked on trace fragments of size 2. It is also important to note that, since it is not conditional upon the agent's being depositing (i.e. it does not use logical implication), this property does *not* describe a transition probability in its strict sense. Instead, it describes the *overall* probability of performing this particular transition and can thus be used to determine the overall probability of an agent gaining energy. For simplicity, we assume that 5 is the maximum level of vision that can be realised technically. The verification results are shown in the last column of Table 9.9. They strengthen the assumption that the scenario with the largest field of vision is the most efficient one since, in this case, agents are most likely to gain energy.

The numbers so far give a strong indication that the probability of grabbing food should be increased. In order to choose the right strategy for achieving this goal, it is essential to *explain* its current level

[8] `GALTL (AAnd (AEq (AAttribute STATE) (ANumVal DEPOSITING) ) (AOr ALast (ANext (AEq (AAttribute STATE) (ANumVal RESTING)))))`

Vision	Pr_1(XI)	Pr_1(XII)	Pr_1(XIII)	Pr_1(XIV)	Pr_1(XV)
1	0.2952	0.0563	0.2694	0.3234	0.0555
2	0.2147	0.1553	0.1668	0.3122	0.1505
3	0.1251	0.2486	0.0810	0.3119	0.2337
4	0.0873	0.2896	0.0470	0.3083	0.2647
5	0.0691	0.3087	0.0330	0.3109	0.2763

TABLE 9.10: Expected individual state distribution

Vision	Pr_1(XI)	Pr_1(XII)	Pr_1(XIII)	Pr_1(XIV)	Pr_1(XV)	Pr_t(I)	Pr_t(IX)
5	0.0925	0.4095	0.0466	0.0828	0.3691	0.0	0.0

TABLE 9.11: Expected state distribution and energy development for $T_r = 1$

first, i.e. to understand *why* it is so low. The intuitive assumption is that an increased field of vision also increases competition among robots which itself increases the probability of agents missing out when trying to grab food. This assumption can be checked by determining the *expected state distribution*, i.e. the *amount of time a robot is expected to spend in each of the states*. The properties are easy to formulate:

$$STATE = SEARCHING \quad \text{(Property XI)}$$

$$STATE = GRABBING \quad \text{(Property XII)}$$

$$STATE = HOMING \quad \text{(Property XIII)}$$

$$STATE = RESTING \quad \text{(Property XIV)}$$

$$STATE = DEPOSITING \quad \text{(Property XV)}$$

The expected state distribution can be obtained by checking all properties above on trace fragments of size 1, i.e. on individual states. This is important since the properties are state properties and, in order to determine their probability, we thus need to sample from the distribution of states. The verification results are shown in Table 9.10 (the Roman literals denote the individual simLTL properties described in this chapter). It becomes apparent that in case of lower vision, a significantly higher proportion of robots spend their time searching and homing (due to timeouts) than in case of higher vision. However, it also becomes apparent, that in case of higher vision, a significantly higher proportion of agents spend their time grabbing. This suggests that grabbing becomes a bottleneck which impedes foraging. Apart from grabbing, in all scenarios, a significant number of agents spend their time resting.

Vision	Pr_1(XI)	Pr_1(XII)	Pr_1(XIII)	Pr_1(XIV)	Pr_1(XV)	Pr_t(I)	Pr_t(IX)
5	0.0923	0.0816	0.0262	0.4135	0.3893	1.0	0.78

TABLE 9.12: Expected state distribution and energy development for $T_g = 1$

We now have two possible directions to improve the overall efficiency of the swarm: we can either try to decrease the time individuals spend for resting or we can try to decrease the time spent for grabbing food items. In order to compare the effect of both changes, we determine again the expected state distribution for each of the two cases. The results are shown in Table 9.11 and 9.12. Reducing the resting time to 1 has the effect of forcing more robots into searching, grabbing and depositing. Likewise, reducing the grabbing time to 1 forces more robots into searching, resting and depositing. Both scenarios only differ with respect to the number of agents grabbing or resting. Taking into account the energy consumption of each agent intuitively suggests that scenario 2 (reduced grabbing time) must be significantly more effective since, in this case, more agents are resting and resting consumes significantly less energy than depositing. This assumption can also be strengthened using MC^{2}MABS by looking at the overall probability of Property I (shown in Column 7) of Tables 9.11 and 9.12. In the case of reduced resting time, the probability of the swarm always having positive energy is 0; in the case of reduced grabbing time, the probability is 1.0. In terms of individual energy levels, individual robots have a probability of always having positive energy of $\approx 78\%$, as shown in Column 8 of Tables 9.11 and 9.12, respectively.

We have now reached a situation in which the overall swarm energy level as well as the majority of all individual energy levels are always positive. However, there is still a significant number of robots ($\approx 22\%$) which eventually run out of energy. In order to find a solution to this problem, we again need to find an *explanation* first, i.e. we first need to understand *why* their energy level falls below zero. It may, for example, be the case, that they run out of energy because they cannot find enough food. They may also run out of energy only in the very beginning of the simulation because it takes some time to find, grab and deposit the first food item. The latter problem could, most likely, be solved by simply increasing the amount of initial energy. The first problem would probably be harder to solve and require re-engineering of the overall energy consumption, e.g. by turning off agents in the resting state.

In order to identify the cause of the problem, we first aim to find out how many agents will eventually run out of energy without any chance to recover by finding food. This can be done by determining the probability of agents to *eventually always* have a negative energy level:

$$\mathbf{FG}(ENERGY \leq 0)^{9} \qquad \text{(Property XVI)}$$

Num. ticks	$1.0 - Pr_t(\mathbf{FG}(ENERGY \geq 0))$
1.000	0.24
500	0.24
100	0.24
50	0.12
20	0.05
10	0.05

TABLE 9.13: Expected individual probability of running out of energy for different simulation lengths

Care needs to be taken when interpreting the verification results for this property. As emphasised in Section 5.2.2, the semantics of nested temporal properties in the presence of finite traces differ significantly from their intuitive meaning. As a consequence, verification results should, in this case, always considered vaguely indicative rather than definite.

The verification of Property XVI returns a probability of just about 1% which indicates that running out of energy seems to be a punctual phenomenon which only occurs at certain points during the simulation; as soon as agents find food, they are able to recover again. We can thus rule out the first hypothesis (bearing in mind the semantics of '**FG**' on finite traces).

The second hypothesis made above stated that agents run out of energy in the very beginning of the simulation because they need too much time (and energy) to find, grab and deposit their first food item. This hypothesis can be tested by repeatedly re-verifying Property XVI on simulations with successively reduced numbers of ticks. The results are shown in Table 9.13. For clarity, the probability has been inverted. The resulting value now describes the expected probability of an individual agent to run out of energy. The numbers show that negative energy levels are reached primarily within the first 100 ticks which indicates that it may, in fact, be an initialisation artefact. A swarm designer can use this insight as a starting point for increasing the initial energy levels, for improving energy intake when consuming a food item or for reducing energy consumption during searching, grabbing, etc. Similar to the previous steps, this process can also be guided by appropriate verification criteria; this, however, is not described in further detail here.

[9] `GALTL (AFinally (AGlobally (ALEq (AAttribute ENERGY) (ANumVal 0))))`

9.6 Summary

This chapter described the application of simLTL-based statistical runtime verification to a real-world scenario from the swarm robotics domain. We showed how MC^{2}MABS can be used to verify properties on different observational levels (micro, meso and macro) and on models with different levels of complexity. The most obvious purpose of verification is to assess the *analytical adequacy* of the model, i.e. the correctness of its internal mechanisms. In a probabilistic setting, potential causal relationships can be studied by analysing the likelihood of a component (e.g. a robot or a group of robots) switching from one state into another. This was illustrated, for example, by ensuring that the transition probabilities determined at runtime correspond with the ones found in the specification. Apart from the confirmation of expected facts, verification is also particularly useful for *revealing* and *quantifying* previously unknown emergent phenomena and their potential causal relationships. Automated quantification can help to make models more transparent, to increase the modeller's understanding of, and, ultimately also his confidence in the dynamics of the model. Examples of this are the calculation of transition probabilities or state distributions and their comparison across configurations which help to explain why certain things happen in the model. And finally, quantification can also help to make models comparable. This was illustrated in Section 9.4 where verification was used to show that the transition probabilities of the microsimulation model and those of the previous equation-based macroscopic model are aligned.

In particular, we provided the following evaluations:

- We showed how statistical runtime verification can be used to internally validate different types of models (macroscopic, microsimulation xand agent-based)
- We showed how simLTL can be used to formulate correctness criteria about a swarm robotic model on different observational levels
- We showed how statistical runtime verification can be used beyond pure safety checking by verifying the correctness of transition probabilities and state distributions
- We showed the practical application of MC^{2}MABS and provided information about the runtime of the verification checks

We have deliberately chosen a technical scenario for this case study. However, it is important to stress, that the aforementioned types of analysis can be equally applied to, for example, agent-based models from the social simulation domain, as illustrated at various points in previous chapters.

Chapter 10

Conclusions

10.1 Introduction

In this thesis, we have described the development of a verification framework for agent-based simulations. Despite impressive advances in combating state space explosion by means of abstraction, reduction or symbolic techniques, conventional model checking still suffers from fundamental complexity issues. Due to their inherent complexity, the verification of agent-based simulations lies beyond the capabilities of existing tools and techniques. The framework described in this thesis aims to combine the ideas of *runtime verification* and *approximate model checking* and unify them in a common conceptual and practical framework. Both ideas are particularly interesting for systems whose complexity prevents them from being verified using conventional methods; we believe that their combination provides a powerful basis for the verification of even large-scale agent-based simulations in a timely manner. As opposed to pure runtime verification, our approach allows for precise quantification of the accuracy with respect to the verification result; as opposed to pure approximate model checking, all checks are performed upon the original system (or a sufficiently representative model thereof).

The framework described in this thesis allows users to formulate their expectations about the desired behaviour of the model in a formal, descriptive and compact way by means of temporal logic; the behaviour of the model itself can be formulated in C++, a powerful high-level programming language. Using a language like C++ for model description instead of a dedicated domain-specific language like, for example, Reactive Modules, ISPL or Promela has two distinct benefits: (i) it helps to keep the

description of the model aligned with its original logic, and (ii) it also allows for 'replaying' existing simulation output, i.e. reading in output that has been produced by a third-party simulation tool such as, for example, NetLogo, and performing verification upon it. In the latter case, verification is effectively decoupled from simulation.

Apart from their computational complexity, agent-based simulations possess a set of characteristics which make them particularly suitable for the application of the aforementioned techniques. First, due to their interdisciplinary nature, agent-based models are typically written in a high-level (and often domain-specific) programming language. Forcing developers to translate their (often complex) logic into a verification-specific modelling language is likely to have a negative impact on the acceptance of the proposed technique; a solution which allows for the verification of the original model is preferable. Second, agent-based simulations are often highly stochastic in nature and individual simulation runs inherently represent random samples of the underlying probability space. The inherent sampling performed through simulation serves as a straightforward starting point for a Monte Carlo-based estimation of the probability of a property. Third, in most cases, agent-based simulations are not safety-critical themselves and instead used for exploration and explanation of the modelled phenomenon; an approximate verification result which can be obtained in reasonable time is therefore mostly perfectly acceptable and favourable over an accurate result which requires a large amount of time.

10.2 Contributions

This thesis makes the following contributions:

Theoretical work: the theoretical part comprises four major contributions:

1. A formalisation of *events* and their relation with *properties* against the background of the underlying sampling space and the corresponding notion of *trace fragments*; we showed that, depending on the interpretation of the sampling space and the subsequent choice of trace fragments, *different levels of granularity* with respect to property formulation can be achieved.
2. The definition of *simLTL*, a linear temporal logic tailored to the formulation of correctness criteria of (potentially large) populations of agents; in order to allow for a finer level of granularity with respect to property formulation, simLTL extends LTL with a distinction

between agent-level and global-level formulae as well as with dedicated selection and quantification operators; in order to allow for the verification of properties on finite traces, the language is given appropriate semantics.

3. The specification of an *evaluation algorithm* which acts as a *monitor* that allows for the verification of simLTL properties on-the-fly, i.e. during the execution of a simulation. By intertwining simulation and property evaluation, the algorithm achieves *laziness* with the overall benefit of reducing the amount of computation necessary to answer a given property to a minimum. Building upon the idea of approximate model checking, simulation and evaluation are executed repeatedly in order to explore the underlying probability space to a measurable extent; as a consequence, the accuracy of the approximate verification results becomes clearly quantifiable and configurable.

4. A description of how *advanced types of analyses* necessary for the verification and validation of agent-based simulations can be performed using the model checking algorithms described. Examples of advanced types of analyses are *correlation analysis*, *conditional analysis*, *causal analysis* or the analysis of the *expected behaviour of a representative agent* (or group).

Implementation: The conceptual ideas described above were all implemented into `MC`^{2}`MABS`, a statistical runtime verification framework for agent-based simulations. `MC`^{2}`MABS` comprises both a *simulator*, which allows for the formulation of arbitrary model logic in a high-level programming language (C++), and a *monitor*, which incorporates the runtime algorithms described above and checks given properties against a set of temporal traces produced by the simulation.

Evaluation: The applicability of the aforementioned concepts and tools was investigated in a range of experiments. First, the performance of `MC`^{2}`MABS` was analysed by investigating the impact of several variables (e.g. population size, formula size or fragment size) on both runtime and memory. Second, `MC`^{2}`MABS` was applied to a larger-scale case study in the area of swarm robotics in order to show its usefulness in a realistic scenario.

The benefits of the developed concepts over existing work can be summarised as follows:

Immediacy with respect to the verification target: Traditional model checkers require a translation of the system logic into a dedicated modelling language which is then used to construct a finite state representation (or an abstraction thereof). We believe that this translation step is only feasible

for comparatively simple systems and would, in the case of agent-based simulations, represent a critical limitation. As a potential solution, we allow for the formulation of the model logic in a general purpose programming language (C++) and define communication between simulation and model checker by means of a strictly defined interface. We believe that this gives a modeller more freedom with respect to the implementation of the model logic and helps to keep the original simulation and the verification model closely aligned.

Expressivity in the property specification language: Current temporal logics are not well-suited for the formulation of properties about large-scale populations of agents. In order to describe correctness criteria for agent-based simulations in a succinct way, issues of granularity and quantification arise. The larger the underlying population gets, the less impact an individual agent has and the more importance is attributed to statements about the entire population or subgroups thereof, about the cardinality of the group of agents satisfying a certain criterion or about the expected behaviour of an average agent within a certain group. Existing specification languages do not currently support the formulation of such properties. With its support for agent and group formulae as well as for selection and quantification, simLTL allows for the formulation of those complex multi-level properties in a compact and descriptive way.

Efficiency with respect to the verification of large-scale systems: State space explosion still represents a critical problem which severely constrains the use of formal model checking for the verification of large-scale software systems. For cases where approximate answers are acceptable, runtime verification can provide an interesting alternative. By intertwining stochastic simulation and verification and thus combining ideas of runtime verification with those of approximate model checking, $\texttt{MC}^2\texttt{MABS}$ is able to achieve a good balance between the accuracy of the verification results and the level of computation necessary. Due to the principle of laziness, only that part of the state space which is necessary for answering a given property needs to be computed by means of simulation; properties whose truth (or falsity) can be determined early can thus be verified in a short amount of time — even for large-scale simulations. The runtime verification framework is easily parametrisable to the circumstances of the verification task. If sufficient resources are available, a high amount of confidence can be achieved efficiently; in all other cases, verification is still perfectly possible, albeit with a lower level of accuracy. Finally, verification is ‘anytime’, i.e. results are available at any point in time.

10.3 Limitations and future work

In addition to its inherently approximate nature, the approach described in this work faces a number of limitations which are briefly summarised below.

- As opposed to existing verification techniques for (more general) multiagent systems, our approach is limited to the verification of the basic temporal behaviour; properties about higher-level notions such as knowledge, beliefs, intentions or goals cannot be evaluated.
- Due to the purely state-based focus, actions are currently not represented explicitly in the specification language. Properties which require the existence of actions, e.g. those about the strategic behaviour of agents and teams, can therefore not be evaluated.
- The semantic model used for verification is currently restricted to the states of the individual agents only which renders the evaluation of properties about environmental dynamics impossible.
- Due to its focus on finite traces, the approach does not allow for the verification of steady-state properties, i.e. properties about the long-run behaviour of the simulation.

In the course of the research project, a number of problems and opportunities for further research have been identified. An overview is given below.

10.3.1 Verifying branching-time properties

Due to its focus on individual traces, the work described in this thesis employs a purely linear notion of time. It would be interesting to extend the approach towards the verification of branching time properties (e.g. CTL or CTL*) in a way similar to that of *on-the-fly model checking* [30]. In a branching time setting, each state may have different possible successor states which are chosen either nondeterministically or randomly based on a probability distribution. Even though individual simulation traces are inherently linear w.r.t. time, the interleaving of simulation and verification built into the algorithms described in Section 6 also allow, in principle, for a more exhaustive exploration of the state space. In this case, the traversal mode becomes important. In case of breadth-first traversal, the space could be explored by means of *repeatedly sampling successor states*. Consider, for example, a state s that has a set of successor states S, each of which can be reached with a particular probability. By repeatedly

performing a state update on s, a sampling of successor states $s' \in S$ could be performed and the transition probabilities between s and $s' \in S$ could be estimated. A recursive application of this process would allow for iterative breadth-first traversal of the state space and the fragment of the state space that needs to be traversed would become a direct function of the nature of the underlying property. In worst case, an exponentially growing number of states would have to be kept in memory. Alternatively, depth-first traversal could be used to explore individual traces and backtrack to previous states in the case of failure. In this case, only the trace currently being analysed would have to be kept in memory, together with information about the possible successor states. This is also the idea that underlies Java Path Finder (JPF) [1] and is exploited, for example, by AJPF [33] (see Section 2.5.1) An important requirement for an existing simulation to be usable in such a scenario is that the `step` function is *side effect-free* in order to be callable multiple times. Although the simulator provided by `MC`^{2}`MABS` could be effectively extended in order to provide this feature, external simulations (which can also be connected to `MC`^{2}`MABS`) cannot be expected to satisfy this requirement. It would thus be necessary to allow for a selection between two different modes of verification — linear time and branching time.

Also related with the verification of branching-time properties is the correspondence between the approach described in this thesis and symbolic model checking as described in Section 6.5 which we aim to investigate further as part of our future work. Due to their compressed representation, symbolic approaches generally allow for the verification of significantly larger systems than their explicit counterparts. For blackbox systems whose transition relation is unknown, symbolic approaches are typically not appropriate. However, the approach described in this thesis could be used to 'learn' the transition relation from a repeatedly executed simulation model as a pre-step towards symbolic verification.

10.3.2 Extension of the logic-based specification language

In its current state, both the syntax and semantics of simLTL are fairly simple and view the evolution of an agent-based simulation as a simple sequence of states. Given the current state of agent-based modelling as an analytical tool, its typical usage in social simulation and the types of analyses typically performed upon such a model, the purely state-based view is largely sufficient; from a technical point of view, agents in agent-based models are mostly reactive and, due to the focus on emergent behaviours, fairly simple in terms of their internal logic. In the case of other types of agent-based systems where the focus is more on problem solving rather than pure representation, the agents' internals become more relevant. In order to reason about those systems, epistemic and strategic versions of temporal logic

have been developed, e.g. the epistemic modal logic S5 [91], ATL [4] and ATEL [231]. The techniques described in this work are not directly applicable to those systems yet, although statistical runtime verification can also be beneficial to combat state space explosion in this case.

As described in Section 5.2.2, the convenience of LTL over has recently been integrated with the expressive power of regular expressions by De Giacomo and Vardi [69]. The resulting logic, LDL_f is strictly more expressive than LTL_f but equally convenient. It would be interesting to consider the usefulness of LDL_f against the background of correctness criteria for agent-based simulations in order to overcome some of the problems with nested temporal operators in the context of finite traces as described in Section 5.2.2.

One interesting future direction, with the overall goal of extending the range of systems verifiable with the runtime verification framework described here, is to associate the idea of agent and simulation traces in this work with the computationally grounded formalism of *interpreted systems* [91]. This would allow for integrating Kripke semantics into the semantic framework of agent and simulation traces used in this work and pave the way for the integration of additional modal, e.g. epistemic, alethic, doxastic or deontic, operators.

10.3.3 Exploring the possibility to analyse steady-state probabilities

Due to its focus on finite traces, runtime verification can, in general, not be used to answer steady-state properties of the underlying system. However, it is well known that, in infinite traces of finite state systems, there is at least one state which is visited twice. As soon as a state which has been visited previously is discovered, the finite trace up to this state can be extended to an infinite trace by following the discovered loop infinitely often. This is the idea that underlies *bounded model checking*, an approach that starts with finite traces which are extended until (i) a *witness*, i.e. a path on which the given property holds, is found, (ii) the problem becomes intractable, or (iii) an upper bound on the number of states has been reached [31]. It would be interesting to apply the same idea to runtime verification and study the usefulness of information about loops for answering properties which would otherwise not be decidable on finite traces.

10.3.4 Improving the efficiency of MC^{2}MABS

As illustrated in Section 8.6, housekeeping tasks — primarily garbage collection and marshalling of data structures — represent an increasingly substantial part of the runtime of MC^{2}MABS as the number of agents in the underlying population grows. The first problem can be addressed by making some of the lazy operations in the Haskell part of the verification framework strict; this might result in increased memory consumption, yet avoid extensive continuous garbage collection. Marshalling of data structures, on the other hand, represents a necessary evil in any cross-language project and cannot be avoided in general; however, by using lightweight data types (e.g. `Int` instead of `String`), the overhead of marshalling can be kept low. Some optimisations have already been exploited in the current version of MC^{2}MABS; as part of the future work, we are planning to explore further opportunities.

An alternative solution, as soon as the logic converges to a stable state, would be to rewrite the verification part of the framework in C++. This would resolve marshalling issues and relax the constraints on exchange data types between the simulation framework and the model checker. Furthermore, against the background of releasing the source code of MC^{2}MABS as open source, C++ is certainly preferable to Haskell from a community perspective. On the other hand, given the strong benefits of lazy evaluation, the comparable performance of compiled Haskell and the significantly higher level of clarity and maintainability of the code, such a translation needs to be carefully considered.

A further important extension of MC^{2}MABS is to exploit the parallelism ubiquitous on all modern hardware platforms. Since both simulation and verification of an individual run represents a self-contained task which can be performed in isolation, significant speedup can be expected by parallelising the execution on multicore platforms and processing multiple simulation runs simultaneously. In essence, the core functionality of MC^{2}MABS can essentially be seen as a single large *MapReduce* operation: during the mapping step, individual workers which perform simulation and subsequent verification are spawned; during the reduction steps, the individual verification results are aggregated into the overall result. Furthermore, it is important to note that quality assurance rarely comprises a single verification property only, but rather consists of a whole set of properties to be evaluated. It is thus important to allow for parallel evaluation of multiple formulae at runtime.

Apart from performance, there are, of course, numerous usability issues that need to be addressed. One potential direction is to transform MC^{2}MABS into an Eclipse plugin which would allow for better editing capabilities, support syntax checking and syntax highlighting, and thus improve usability significantly.

10.3.5 Reduction of sample size necessary for verification

According to the idea of approximate model checking, a sufficiently high number of simulation runs need to be performed in order to achieve the desired level of accuracy [117]. Due to the exponential relationship between the number of sample paths and the level of accuracy, the latter will, in most cases, remain limited to values between 95 and 99%. Whereas, in social science, 95% accuracy is typically seen as sufficient, this is clearly not the case in the area of safety-critical systems. The functional safety standard IEC 61508, for example, defines four *safety integrity levels (SIL)*; SIL 1 (the lowest level) requires an average failure probability of less than 10^{-1} in *low demand mode*; SIL 4 (the highest level) requires an average failure probability of less than 10^{-4} in low demand mode. In *high demand* or *continuous operation mode*, the probability required by SIL 1 falls to $< 10^{-6}$ and the probability by SIL 4 falls to $< 10^{-8}$ [27]. Those levels of accuracy are clearly hard to achieve using the approach described in Section 6.4.

Nevertheless — as mentioned in the Introduction — agent-based simulations are increasingly being used to model safety-critical systems which demands higher levels of accuracy in terms of verification. One potential way to address this problem is to employ *rare event sampling* methods from statistics, for example *stratified sampling*. Similar work has been done in the area of software testing and debugging [53, 54]. We plan to investigate the applicability of those methods to approximate online runtime verification as part of our future work.

10.3.6 Intra-run causal analysis

So far, causal analysis has been touched upon only very superficially against the background of initial conditions (see Section 7.6). In addition to that, it may also be useful to look for causal relationships *within a single trace*, i.e. once a property has been violated and a counterexample has been created. In this case, causal analysis can help to determine the factors within the model which contributed to the violation of the property. Consider, for example, a property which states that, all agents will eventually be infected. In cases where the property is violated, it might be useful to determine causes within the population. For example, we may want to ask *what-if* questions such as "Would the property still have been violated if agent X had not been infected at time step t?" Those types of question are, for example, relevant for the study of diffusion dynamics in social networks [74].

Causality within counterexamples resulting from model checking has been investigated before [26]. We are planning to further extend our research into this direction and assess the potential of intra-run causal analysis for the explanation of phenomena within agent-based simulations.

10.4 Concluding remarks

Within the last twenty years, multiagent technology has advanced significantly and has shown great power for the development of complex and large-scale distributed systems comprising autonomous, adaptive and heterogeneous agents. Agent-based simulation as one of the many application areas of multiagent technology has rapidly emerged as a powerful alternative to conventional analytical techniques which increasingly struggle with the growing complexity and unpredictability of the world, both in a technical and in a social context. Rather than for the purpose of short-term prediction (which straightforward machine learning may well be more suitable for), agent-based simulation has shown to be of particular use for *explanatory* purposes in complex environments, for the understanding of mechanisms and their interplay, for the detection of causal relationships and the study of emergent phenomena.

However, in order for agent-based simulation to mature and find its application beyond pure academic research (which is still its primary domain), a more rigorous approach to the construction and analysis of simulations is urgently required. Ad-hoc modelling which has been the predominant approach to date is no longer an option if future models are to be significantly more complex, more robust and trustworthy enough to be used as decision support tools for policy making and analysis. In terms of their engineering, simulations should not be treated differently than other types of software; structured approaches to 'simulation engineering' are indispensable if agent-based modelling is to extend beyond mere prototyping and illustration.

The work described in this thesis aims to make a small contribution to this field by bridging the gap between the area of formal software engineering — particularly verification — and agent-based modelling, two fields which have been largely decoupled to date. In designing the verification approach, we aimed to be as general as possible in order not to limit its applicability to a particular modelling domain and to make it available to a wide and diverse range of users with different backgrounds. We hope that this will help to bring more formal rigour into this currently largely empirical field in order to allow for the engineering of more complex and meaningful simulation models. We also hope that more computer

scientists will turn their attention to this exciting field and contribute their knowledge to the development of general tools and methods. Viewing agent-based modelling not only from an analytical but also from an engineering perspective will make an important difference and help to turn it into an even more powerful and intellectually stimulating technique which has the potential of significantly enhancing our understanding of the world that we inhabit.

Appendix A

The Z Notation

The following glossary is adapted from the book "The Way of Z" by Jonathan Jacky [132].

Names

a, b	identifiers
d, e	declarations (e.g., $a : A;\ b, \cdots : B \ldots$)
f, g	functions
m, n	numbers
p, q	predicates
s, t	sequences
x, y	expressions
A, B	sets
C, D	bags
Q, R	relations
S, T	schemas
X	schema text (e.g., d, $d \mid p$, or S)

Definitions

$a == x$	Abbreviation definition

$T ::= c \mid d\langle\langle U \rangle\rangle$	Free type definition
$[a]$	Introduction of a given set (or $[a, \dots]$)
$a\_$	Prefix operator
$\_a$	Postfix operator
$\_a\_$	Infix operator

Logic

$true$	Logical true constant
$false$	Logical false constant
$\neg\ p$	Logical negation, *not*
$p \land q$	Logical conjunction, *and*
$p \lor q$	Logical disjunction, *or*
$p \Rightarrow q$	Logical implication
$p \Leftrightarrow q$	Logical equivalence
$\forall X \bullet q$	Universal quantification
$\exists X \bullet q$	Existential quantification
(**let** $a == x;\ \dots \bullet p$)	Local definition

Sets and expressions

$x = y$	Equality
$x \neq y$	Inequality
$x \in A$	Set membership
$x \notin A$	Non-membership
$\varnothing$	Empty set
$A \subseteq B$	Set inclusion
$A \subset B$	Strict set inclusion
$\{x, y, \dots\}$	Set display
$\{\ X \bullet x\ \}$	Set comprehension
$(\lambda X \bullet x)$	Lambda expression
(**let** $a == x;\ \dots \bullet y$)	Local definition
if p **then** x **else** y	Conditional expression

$(x, y, \dots)$	Tuple
(x, y)	Pair
$A \times B \times \dots$	Cartesian product
$\mathbb{F}\, A$	Finite set
$\mathbb{P}\, A$	Power set
$A \cap B$	Set intersection
$A \cup B$	Set union
$A \setminus B$	Set difference
$x.1$	First element of an ordered pair
$x.2$	Second element of an ordered pair
$\#A$	Number of elements in a set

Relations

$A \leftrightarrow B$	Binary relation ($\mathbb{P}(A \times B)$)
$a \mapsto b$	Maplet ((a, b))
dom R	Domain of a relation
ran R	Range of a relation
$Q \fatsemi R$	Forward relational composition
$Q \circ R$	Backward relational composition ($R \fatsemi Q$)
$A \lhd R$	Domain restriction
$A \lhd\!\!\!\!- R$	Domain anti-restriction
$A \rhd R$	Range restriction
$A \rhd\!\!\!\!- R$	Range anti-restriction
$R(\!\mid A \mid\!)$	Relational image
$R^{\sim}$	Inverse of relation
R^{+}	Transitive closure
$Q \oplus R$	Relational overriding
$a \mathrel{\underline{\mathrm{R}}} b$	Infix relation

Functions

$A \pfun B$	Partial functions

$A \to B$	Total functions
$A \rightarrowtail B$	Partial injections
$A \rightarrowtail B$	Total injections
$A \rightarrowtail\!\!\!\!\!\rightarrow B$	Bijections
$f\ x$	Function application (or $f(x)$)

Numbers

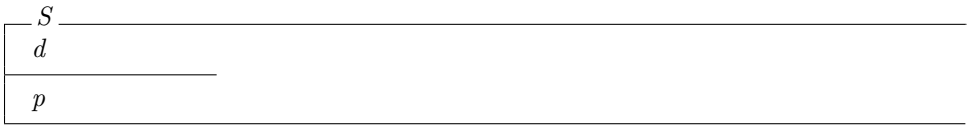
$\mathbb{Z}$	Set of integers
$\mathbb{N}$	Set of natural numbers $\{0, 1, 2, \dots\}$
$\mathbb{N}_1$	Set of strictly positive numbers $\{1, 2, \dots\}$
$\mathbb{R}$	Set of real numbers
$m + n$	Addition
$m - n$	Subtraction
$m * n$	Multiplication
$m \operatorname{div} n$	Division
$m \bmod n$	Remainder (modulus)
$m \leq n$	Less than or equal
$m < n$	Less than
$m \geq n$	Greater than or equal
$m > n$	Greater than
$m \mathinner{..} n$	Number range
$min\ A$	Minimum of a set of numbers
$max\ A$	Maximum of a set of numbers

Sequences

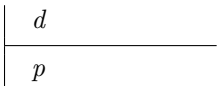
seq A	Set of finite sequences
$\text{seq}_1\ A$	Set of non-empty finite sequences
iseq A	Set of finite injective sequences
$\langle\rangle$	Empty sequence
$\langle x, y, \dots \rangle$	Sequence display
$s \frown t$	Sequence concatenation

$head\ s$	First element of a sequence
$tail\ s$	All but the head element of a sequence
$last\ s$	Last element of a sequence
$front\ s$	All but the last element of a sequence
s in t	Sequence segment relation

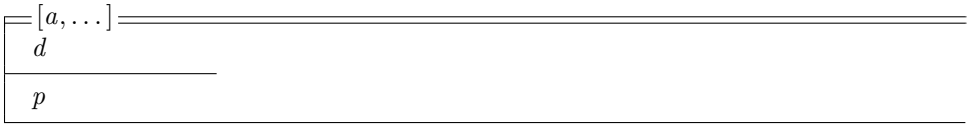
Schema



Axiomatic definition



Generic definition



Schema calculus

$S == [\, X \,]$	Horizontal schema
$[\, T;\ \ldots \mid \ldots \,]$	Schema inclusion
$z.a$	Component selection (given $z : S$)
θS	Binding
$\neg\ S$	Schema negation
$S \land T$	Schema conjunction
$S \lor T$	Schema disjunction

$S \fatsemi T$	Schema composition
$S \gg T$	Schema piping

Conventions

$a?$	Input to an operation
$a!$	Output from an operation
a	State component before an operation
a'	State component after an operation
S	State schema before an operation
S'	State schema after an operation
ΔS	Change of state
ΞS	No change of state

Bibliography

[1] Java Path Finder. http://javapathfinder.sourceforge.net/. Last accessed: 05/2011.

[2] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to AspectJ. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*, pages 345–364, 2005.

[3] R. Alur and T. A. Henzinger. A really temporal logic. *Journal of the ACM*, 41(1):181–203, 1994.

[4] R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time temporal logic. *Journal of the ACM (JACM)*, 49(5):672–713, 2002.

[5] J. R. Anderson, M. D. Byrne, S. Douglass, C. Lebiere, and Y. Qin. An integrated theory of the mind. *Psychological Review*, 111(4):1036–1050, 2004.

[6] W. B. Arthur. Out-of-equilibrium economics and agent-based modeling. In L. Tesfatsion and K. Judd, editors, *Handbook of Computational Economics*, volume 2, chapter 32, pages 1551 – 1564. Elsevier, 2006.

[7] R. Axelrod. *The Evolution of Cooperation*. Basic Books, 1984.

[8] R. Axelrod. An evolutionary approach to norms. *American Political Science Review*, 80(4):1095–1111, 1986.

[9] R. Axelrod. Advancing the art of simulation in the social sciences. In J. P. Rennard, editor, *Handbook of Research on Nature Inspired Computing for Economy and Management*, volume 2. Hersey, PA, 2006.

[10] R. Axtell, R. Axelrod, J. M. Epstein, and M. D. Cohen. Aligning simulation models: A case study and results. *Computational & Mathematical Organization Theory*, 1(2):123–141, 1996.

[11] F. Baader, A. Bauer, and M. Lippmann. Runtime verification using a temporal description logic. In S. Ghilardi and R. Sebastiani, editors, *Proceedings of the 7th International Conference on Frontiers of Combining Systems*, FroCoS'09, pages 149–164. Springer, 2009.

[12] C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[13] N. Bakar and A. Selamat. Runtime verification of multi-agent systems interaction quality. In A. Selamat, N. Nguyen, and H. Haron, editors, *Intelligent Information and Database Systems*, volume 7802 of *Lecture Notes in Computer Science*, pages 435–444. Springer, 2013.

[14] O. Balci. Verification, validation, and accreditation. In *Proceedings of the 30th Winter Simulation Conference (WSC'98)*, pages 41–4, 1998.

[15] O. Balci. Verification, validation, and certification of modeling and simulation applications. In *Proceedings of the 35th Winter Simulation Conference (WSC'03)*, pages 150–158, 2003.

[16] P. Ballarini, M. Fisher, and M. Wooldridge. Uncertain agent verification through probabilistic model-checking. In M. Barley, H. Mouratidis, A. Unruh, D. Spears, P. Scerri, and F. Massacci, editors, *Safety and Security in Multiagent Systems*, volume 4324 of *Lecture Notes in Computer Science*, pages 162–174. Springer, 2009.

[17] S. Banerjee and M. E. Moses. A hybrid agent based and differential equation model of body size effects on pathogen replication and immune system response. In P. S. Andrews, J. Timmis, N. D. L. Owens, U. Aickelin, E. Hart, A. Hone, and A. M. Tyrrell, editors, *Artificial Immune Systems*, volume 5666 of *Lecture Notes in Computer Science*, pages 14–18. Springer, 2009.

[18] J. Banks and J. S. Carson. Introduction to discrete-event simulation. In *Proceedings of the 18th Winter Simulation Conference (WSC'86)*, pages 17–23, 1986.

[19] Y. Bar-Yam. A mathematical theory of strong emergence using multiscale variety. *Complexity*, 9:15–24, July 2004.

[20] H. Barringer, D. Rydeheard, and K. Havelund. Rule systems for run-time monitoring: From Eagle to RuleR. In O. Sokolsky and S. Taşıran, editors, *Runtime Verification*, volume 4839 of *Lecture Notes in Computer Science*, pages 111–125. Springer, 2007.

[21] D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim. Jass – Java with assertions. *Electronic Notes in Theoretical Computer Science*, 55(2):103 – 117, 2001.

[22] S. Baswana, R. Hariharan, and S. Sen. Improved decremental algorithms for maintaining transitive closure and all-pairs shortest paths. In *Proceedings of the 34 annual ACM symposium on the Theory of Computing (STOC '02)*, pages 117–123, New York, NY, USA, 2002. ACM.

[23] A. Bauer, M. Leucker, and C. Schallhart. Monitoring of real-time properties. In S. Arun-Kumar and N. Garg, editors, *Foundations of Software Technology and Theoretical Computer Science (FSTTCS '06)*, volume 4337 of *Lecture Notes in Computer Science*, pages 260–272. Springer, 2006.

[24] A. Bauer, M. Leucker, and C. Schallhart. Comparing LTL semantics for runtime verification. *Journal of Logic and Computation*, 20(3):651–674, June 2010.

[25] M. A. Bedau. Weak emergence. *Philosophical Perspectives*, 11:375–399, 1997.

[26] I. Beer, S. Ben-David, H. Chockler, A. Orni, and R. Trefler. Explaining counterexamples using causality. *Formal Methods in System Design*, 40(1):20–40, 2012.

[27] R. Bell. Introduction to IEC 61508. In T. Cant, editor, *Proceedings of the 10th Australian Workshop on Safety Critical Systems and Software (SCS '05)*, pages 3–12, Darlinghurst, Australia, 2006. Australian Computer Society, Inc.

[28] G. Beni. From swarm intelligence to swarm robotics. In E. Şahin and W. M. Spears, editors, *Swarm Robotics*, pages 1–9. Springer, 2005.

[29] G. K. Bharathy and B. Silverman. Validating agent based social systems models. In *Proceedings of the Winter Simulation Conference*, pages 441–453. Winter Simulation Conference, 2010.

[30] G. Bhat, R. Cleaveland, and O. Grumberg. Efficient on-the-fly model checking for CTL. In *Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science (LICS '95)*, pages 388–397, 1995.

[31] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58:117–148, 2003.

[32] E. Bonabeau, M. Dorigo, and G. Theraulaz. *Swarm Intelligence*. Oxford, 1999.

[33] R. H. Bordini, L. A. Dennis, B. Farwer, and M. Fisher. Automated verification of multi-agent programs. In P. I. A. Ireland and W. Visser, editors, *Proceedings of the 23rd International Conference on Automated Software Engineering (ASE'08)*, pages 69–78, Washington, DC, USA, 2008. IEEE Computer Society.

[34] R. H. Bordini, M. Fisher, W. Visser, and M. Wooldridge. Verifiable multi-agent programs. In M. Dastani, J. Dix, and A. El Fallah-Seghrouchni, editors, *Programming Multi-Agent Systems*, volume 3067 of *Lecture Notes in Computer Science*, pages 72–89. Springer, 2004.

[35] R. H. Bordini, M. Fisher, W. Visser, and M. Wooldridge. Verifying multi-agent programs by model checking. *Autonomous Agents and Multi-Agent Systems*, 12:239–256, March 2006.

[36] R. H. Bordini and J. F. Hübner. Agent-based simulation using bdi programming in jason. In A. Uhrmacher and D. Weyns, editors, *Multi-Agent Systems: Simulation and Applications*, pages 451–471. Citeseer, 2009.

[37] A. Borshchev and A. Filippov. From system dynamics and discrete event to practical agent based modeling: reasons, techniques, tools. In *Proceedings of the 22nd International Conference of the System Dynamics Society*, number 22, 2004.

[38] T. Bosse and N. Mogles. Comparing modelling approaches in aviation safety. In R. Curran, editor, *Proceedings of the 4th International Air Transport and Operations Symposium (ATOS '13), Toulouse, France*, 2013.

[39] S. Bouarfa, H. Blom, R. Curran, and M. Everdij. Agent-based modeling and simulation of emergent behavior in air transportation. *Complex Adaptive Systems Modeling*, 1(1):1–26, 2013.

[40] B. Bruegge, T. Gottschalk, and B. Luo. A framework for dynamic program analyzers. In A. Paepcke, editor, *Proceedings of the 8th Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '93, pages 65–82, New York, NY, USA, 1993. ACM.

[41] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98:142–170, June 1992.

[42] A. Caballero, J. Botía, and A. Gómez-Skarmeta. Using cognitive agents in social simulations. *Engineering Applications of Artificial Intelligence*, 24(7):1098–1109, 2011.

[43] I. Cakirlar, Ö. Gürcan, O. Dikenelli, and S. Bora. RatKit: A repeatable automated testing toolkit for agent-based modeling and simulation. In *Proceedings of the 15th International Workshop on Multi-Agent-Based Simulation (MABS '14)*, 2014.

[44] D. T. Campbell. Downward causation in hierarchically organised biological systems. In F. Ayala and T. Dobzhansky, editors, *Studies in the philosophy of biology: Reduction and related problems*, pages 179–186. Macmillan, 1974.

[45] Y. U. Cao, A. S. Fukunaga, and A. Kahng. Cooperative mobile robotics: Antecedents and directions. *Autonomous Robots*, 4(1):7–27, Mar. 1997.

[46] J. B. Cardoso, J. R. de Almeida, J. M. Dias, and P. G. Coelho. Structural reliability analysis using Monte Carlo simulation and neural networks. *Advances in Engineering Software*, 39:505–513, June 2008.

[47] G. Caron-Lormier, R. W. Humphry, D. A. Bohan, C. Hawes, and P. Thorbek. Asynchronous and synchronous updating in individual-based models. *Ecological Modelling*, 212(3-4):522 – 527, 2008.

[48] C. J. Castle and A. T. Crooks. Principles and concepts of agent-based modelling for developing geospatial simulations. Working paper, Centre for Advanced Spatial Analysis, UCL (University College London), 2006.

[49] F. E. Cellier and E. Kofman. *Continuous System Simulation*. Springer, Secaucus, NJ, USA, 2006.

[50] W. K. V. Chan, Y. J. Son, and C. M. Macal. Agent-based simulation tutorial — simulation of emergent behavior and differences between agent-based simulation and discrete-event simulation. In B. Johansson, S. Jain, J. Montoya-Torres, J. Hugan, and E. Yücesan, editors, *Proceedings of the 2010 Winter Simulation Conference*, pages 135–150. WSC, 2010.

[51] C.-C. Chen, S. B. Nagl, and C. D. Clack. Specifying, detecting and analysing emergent behaviours in multi-level agent-based simulations. In G. A. Wainer, editor, *Proceedings of the 2007 Summer Simulation Conference (SCSC '07)*, pages 969–976, San Diego, CA, USA, 2007. Society for Computer Simulation International.

[52] S.-H. Chen, C.-L. Chang, and Y.-R. Du. Agent-based economic models and econometrics. *The Knowledge Engineering Review*, 27:187–219, 6 2012.

[53] H. Chockler, E. Farchi, B. Godlin, and S. Novikov. Cross-entropy based testing. In *Formal Methods in Computer Aided Design*, pages 101–108. IEEE, 2007.

[54] H. Chockler, E. Farchi, B. Godlin, and S. Novikov. Cross-entropy-based replay of concurrent programs. In *Fundamental Approaches to Software Engineering*, pages 201–215. Springer, 2009.

[55] S. E. Chodrow and M. G. Gouda. The Sentry system. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, pages 230–237. IEEE, 1992.

[56] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer (STTT)*, 2:410–425, 2000.

[57] E. Clarke. Model checking. In S. Ramesh and G. Sivakumar, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 1346 of *Lecture Notes in Computer Science*, pages 54–56. Springer, 1997.

[58] E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19:7–34, July 2001.

[59] E. Clarke, E. Emerson, S. Jha, and A. Sistla. Symmetry reductions in model checking. In A. Hu and M. Vardi, editors, *Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 147–158. Springer, 1998.

[60] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Progress on the state explosion problem in model checking. In R. Wilhelm, editor, *Informatics*, volume 2000 of *Lecture Notes in Computer Science*, pages 176–194. Springer, 2001.

[61] E. Clarke, O. Grumberg, M. Minea, and D. Peled. State space reduction using partial order techniques. *International Journal on Software Tools for Technology Transfer (STTT)*, 2:279–287, 1999.

[62] E. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[63] N. Collier. Repast: An extensible framework for agent simulation. *Natural Resources and Environmental Issues*, 8, 2001.

[64] J. C. Corbett, M. B. Dwyer, J. Hatcliff, and Robby. Bandera: a source-level interface for model checking Java programs. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE '00)*, pages 762–765, New York, NY, USA, 2000.

[65] D. Cornforth, D. G. Green, D. Newth, and M. Kirley. Do artificial ants march in step? Ordered asynchronous processes and modularity in biological systems. In R. Standish, M. Bedau, and H. A. Abbass, editors, *Proceedings of the 8th International Conference on Artificial Life (ICAL'03)*, pages 28–32, Cambridge, MA, USA, 2003. MIT Press.

[66] T. Crane. Causality. In A. C. Grayling, editor, *Philosophy: A Guide Through the Subject*. Oxford University Press, 1998.

[67] N. David. Validation and verification in social simulation: Patterns and clarification of terminology. In F. Squazzoni, editor, *Epistemological Aspects of Computer Simulation in the Social Sciences*, volume 5466 of *Lecture Notes in Computer Science*, pages 117–129. Springer, 2009.

[68] P. Davidsson. Agent based social simulation: a computer science view. *Journal of Artificial Societies and Social Simulation*, 5:1, 2002.

[69] G. De Giacomo and M. Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI '13)*, pages 854–860. AAAI Press, 2013.

[70] T. De Wolf, T. Holvoet, and G. Samaey. Development of self-organising emergent applications with simulation-based numerical analysis. In S. A. Brueckner, D. M. S. G., D. Hales, and F. Zambonelli, editors, *Engineering Self-Organising Systems*, volume 3910 of *Lecture Notes in Computer Science*, pages 138–152. Springer, 2006.

[71] M. I. Dekhtyar, A. J. Dikovsky, and M. K. Valiev. Temporal verification of probabilistic multiagent systems. In A. Avron, N. Dershowitz, and A. Rabinovich, editors, *Pillars of Computer Science*, pages 256–265. Springer, 2008.

[72] C. Delgado and M. Benevides. Verification of epistemic properties in probabilistic multi-agent systems. In L. Braubach, W. van der Hoek, P. Petta, and A. Pokahr, editors, *Proceedings of the 7th German Conference on Multiagent System Technologies (MATES'09)*, pages 16–28. Springer, 2009.

[73] N. Delgado, A. Gates, and S. Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Transactions on Software Engineering*, 30(12):859–872, Dec 2004.

[74] S. A. Delre, W. Jager, T. H. A. Bijmolt, and M. A. Janssen. Will it spread or not? the effects of social influences and network topology on innovation diffusion. *Journal of Product Innovation Management*, 27(2):267–282, 2010.

[75] S. Demri. Linear-time temporal logics with Presburger constraints: an overview. *Journal of Applied Non-Classical Logics*, 16(3–4):311–347, 2006.

[76] S. Demri and D. D'souza. An automata-theoretic approach to Constraint LTL. In M. Agrawal and A. Seth, editors, *Foundations of Software Technology and Theoretical Computer Science*, volume 2556 of *Lecture Notes in Computer Science*, pages 121–132. Springer, 2002.

[77] J. B. Detemple, R. Garcia, and M. Rindisbacher. A Monte Carlo method for optimal portfolios. *The Journal of Finance*, 58(1):401–446, 2003.

[78] D. A. Dickey and W. A. Fuller. Distribution of the estimators for autoregressive time series with a unit root. *Journal of the American Statistical Association*, 74(366):pp. 427–431, 1979.

[79] M. d'Inverno and M. Luck. *Understanding Agent Systems*. Springer, 2004.

[80] R. Donaldson and N. Gilbert. A Monte Carlo model checker for probabilistic LTL with numerical constraints. Technical Report 282, Department of Computing Science, University of Glasgow, 2008.

[81] D. Edgington. Conditionals. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Winter 2008 edition, 2008.

[82] E. Eells. *Probabilistic Causality*. Cambridge University Press, 1991.

[83] C. Eisner. PSL for runtime verification: Theory and practice. In O. Sokolsky and S. Taşıran, editors, *Proceedings of the 7th International Conference on Runtime Verification (RV'07)*, pages 1–8. Springer, 2007.

[84] C. Eisner, D. Fisman, J. Havlicek, Y. Lustig, A. McIsaac, and D. Van Campenhout. Reasoning with temporal logic on truncated paths. In W. A. Hunt and F. Somenzi, editors, *Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 27–39. Springer, 2003.

[85] C. Elsenbroich. Explanation in agent-based modelling: Functions, causality or mechanisms? *Journal of Artificial Societies and Social Simulation*, 15(3), June 2012.

[86] J. B. Elsner. Granger causality and atlantic hurricanes. *Tellus A*, 59(4):476–485, 2007.

[87] J. Epstein. Why model? *Journal of Artificial Societies and Social Simulation*, 11(4), 2008.

[88] J. M. Epstein. Remarks on the foundations of agent-based generative social science. In L. Tesfatsion and K. L. Judd, editors, *Handbook of computational economics*, volume 2, pages 1585–1604. Elsevier, 2006.

[89] J. M. Epstein and R. L. Axtell. *Growing Artificial Societies: Social Science from the Bottom Up*. MIT Press, June 1996.

[90] F. Fages and A. Rizk. On the analysis of numerical data time series in temporal logic. In M. Calder and S. Gilmore, editors, *Proceedings of the 5th International Conference on Computational Methods in Systems Biology (CMSB'07)*, pages 48–63. Springer, 2007.

[91] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning about Knowledge*. MIT Press, Cambridge, 1995.

[92] H. Fecher, M. Leucker, and V. Wolf. Don't know in probabilistic systems. In *Model Checking Software*, pages 71–88. Springer, 2006.

[93] J. Ferziger and M. Peric. *Computational Methods for Fluid Dynamics*. Springer, third edition, December 2001.

[94] C. Fischer. CSP-OZ: a combination of Object-Z and CSP. In H. Bowman and J. Derrick, editors, *Proceedings of the International Workshop on Formal Methods for Open Object-Based Distributed Systems*, pages 423–438, London, UK, 1997. Chapman & Hall, Ltd.

[95] S. Franklin and A. Graesser. Is it an agent, or just a program?: A taxonomy for autonomous agents. In J. Müller, M. Wooldridge, and N. Jennings, editors, *Proceedings of the 3rd International Workshop on Agent Theories, Architectures, and Languages (ATAL'96)*, volume 1193 of *Lecture Notes in Computer Science*, pages 21–35. Springer, 1997.

[96] D. Gabbay. The declarative past and imperative future. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Temporal Logic in Specification*, volume 398 of *Lecture Notes in Computer Science*, pages 409–448. Springer, 1989.

[97] J. M. Galán, L. R. Izquierdo, S. S. Izquierdo, J. I. Santos, R. del Olmo, A. López-Paredes, and B. Edmonds. Errors and artefacts in agent-based modelling. *Journal of Artificial Societies and Social Simulation*, 12(1), 2009.

[98] L. Gasser and J.-P. Briot. Distributed Artificial Intelligence: Theory and Practice. chapter Object-based Concurrent Programming and Distributed Artificial Intelligence, pages 81–107. Kluwer Academic Publishers, Norwell, MA, USA, 1992.

[99] P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01)*, pages 53–65, London, UK, 2001. Springer.

[100] M. A. C. Gatti and A. von Staa. Testing & debugging multi-agent systems: a state of the art report. Technical report, Departamento de Informatica, Pontifical Catholic University of Rio de Janeiro, 2006.

[101] M. P. Georgeff, B. Pell, M. E. Pollack, M. Tambe, and M. Wooldridge. The belief-desire-intention model of agency. In *Proceedings of the 5th International Workshop on Intelligent Agents V, Agent Theories, Architectures, and Languages*, ATAL '98, pages 1–10, London, UK, 1999. Springer.

[102] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In P. Dembinski and M. Sredniawa, editors, *Proceedings of the 15th IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV*, pages 3–18, London, UK, 1996. Chapman & Hall, Ltd.

[103] J. Geweke. Monte carlo simulation and numerical integration. volume 1 of *Handbook of Computational Economics*, chapter 15, pages 731 – 800. Elsevier, 1996.

[104] N. Gilbert. Varieties of emergence. In C. Macal and D. Sallach, editors, *Social Agents: Ecology, Exchange, and Evolution*, pages 41–56. University of Chicago and Argonne National Laboratory, 2002.

[105] N. Gilbert. Agent-based social simulation: Dealing with complexity. Working paper, Centre for Research on Social Simulation, University of Surrey, 2004.

[106] N. Gilbert. When does social simulation need cognitive models. *Cognition and Multi-Agent Interaction*, page 428, 2006.

[107] N. Gilbert. *Agent-based models*. Quantitative applications in the social sciences. Sage, Los Angeles, CA, 2008.

[108] N. Gilbert and R. Conte. *Artificial Societies: The Computer Simulation of Social Life*. Taylor & Francis, Inc., Bristol, PA, USA, 1995.

[109] N. Gilbert and K. G. Troitzsch. *Simulation for the Social Scientist*. Open University Press, first edition, 1999.

[110] N. Gilbert and K. G. Troitzsch. *Simulation for the Social Scientist*. Open University Press, second edition, 2005.

[111] D. Goldsman. Introduction to simulation. In *Proceedings of the 39th Winter Simulation Conference (WSC '07)*, pages 26–37, 2007.

[112] R. Grosu and S. A. Smolka. Monte Carlo model checking. In N. Halbwachs and L. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *Lecture Notes in Computer Science*, pages 271–286. Springer, 2005.

[113] J. Y. Halpern and R. Fagin. A formal model of knowledge, action, and communication in distributed systems: Preliminary report. In *Proceedings of the 4th Annual ACM Symposium on Principles of Distributed Computing (PODC '85)*, pages 224–236, New York, NY, USA, 1985. ACM.

[114] H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6(5):512–535, 1994.

[115] B. Heath, R. Hill, and F. Ciarallo. A survey of agent-based modeling practices (January 1998 to July 2008). *Journal of Artificial Societies and Social Simulation*, 12(4):9, 2009.

[116] P. Hedström and P. Ylikoski. Causal mechanisms in the social sciences. *Annual Review of Sociology*, 36(1):49–67, June 2010.

[117] T. Hérault, R. Lassaigne, F. Magniette, and S. Peyronnet. Approximate probabilistic model checking. In *Proceedings of the 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'04)*, volume 2937 of *Lecture Notes in Computer Science*, pages 307–329. Springer, 2004.

[118] B. Herd, S. Miles, P. McBurney, and M. Luck. Reachability analysis for agent-based simulations. In *Proceedings of the 1st International Workshop on Verification and Validation of Multi-agent models for complex systems (V2CS '11)*, Paris, France, 2011.

[119] B. Herd, S. Miles, P. McBurney, and M. Luck. Compositional transient analysis for agent-based simulations. *Studia Informatica Universalis*, 10(3):87–118, 2012.

[120] B. Herd, S. Miles, P. McBurney, and M. Luck. Verification and validation of agent-based simulations using approximate model checking. In *Proceedings of the 14th International Workshop on Multi-Agent-Based Simulation (MABS '13)*, St Paul, MN, USA, 2013.

[121] B. Herd, S. Miles, P. McBurney, and M. Luck. Verification and validation of agent-based simulations using approximate model checking. In S. J. Alam and H. V. D. Parunak, editors, *Multi-Agent-Based Simulation XIV*, Lecture Notes in Computer Science, pages 53–70. Springer, 2014.

[122] J. Hillston. *A compositional approach to performance modelling*. Cambridge University Press, New York, NY, USA, 1996.

[123] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. PRISM: A tool for automatic verification of probabilistic systems. In H. Hermanns and J. Palsberg, editors, *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06)*, volume 3920 of *Lecture Notes in Computer Science*, pages 441–444. Springer, 2006.

[124] C. Hitchcock. Probabilistic causation. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Winter 201 edition, 2012.

[125] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21:666–677, August 1978.

[126] C. A. R. Hoare. *Communicating Sequential Processes*, volume 178. Prentice Hall International (UK) Ltd., 1985.

[127] W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963.

[128] G. Holzmann. *The Spin model checker: primer and reference manual*. Addison-Wesley Professional, first edition, 2003.

[129] D. Hume. *A Treatise of Human Nature*. Courier Dover Publications, 2003.

[130] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2004.

[131] L. R. Izquierdo, S. S. Izquierdo, J. M. Galán, and J. I. Santos. Techniques to understand computer simulations: Markov chain analysis. *Journal of Artificial Societies and Social Simulation*, 12(1):6, 2009.

[132] J. Jacky. *The Way of Z: Practical Programming with Formal Methods*. Cambridge University Press, 1997.

[133] W. D. Kelton. Statistical issues in simulation. In *Proceedings of the 26th Winter Simulation Conference (WSC'96)*, pages 47–54, 1996.

[134] W. D. Kelton. Designing simulation experiments. In *Proceedings of the 31st Winter Simulation Conference (WSC'99)*, pages 33–38, 1999.

[135] W. D. Kelton and R. R. Barton. Experimental design for simulation. In *Proceedings of the 34th Winter Simulation Conference (WSC '03)*, pages 59–65, 2003.

[136] Y. Kesten, A. Pnueli, and L. Raviv. Algorithmic verification of linear temporal logic specifications. In K. Larsen, S. Skyum, and G. Winskel, editors, *Automata, Languages and Programming*, volume 1443 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 1998.

[137] M. Kim, M. Viswanathan, H. Ben-Abdallah, S. Kannan, I. Lee, and O. Sokolsky. Formally specified monitoring of temporal properties. In *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, pages 114–122. IEEE, 1999.

[138] M. Kiran, P. Richmond, M. Holcombe, L. S. Chin, D. Worth, and C. Greenough. FLAME: Simulating large populations of agents on parallel hardware architectures. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems (AAMAS'10)*, pages 1633–1636, Richland, SC, 2010.

[139] J. P. C. Kleijnen. Validation of models: statistical techniques and data availability. In *Proceedings of the 31st Winter Simulation Conference (WSC'99)*, pages 647–654, 1999.

[140] S. Kleinberg and B. Mishra. The temporal logic of causal structures. In *Proceedings of the 25th Conference on Uncertainty in Artificial Intelligence*, pages 303–312. AUAI Press, 2009.

[141] S. Konur, C. Dixon, and M. Fisher. Formal verification of probabilistic swarm behaviours. In M. Dorigo, M. Birattari, G. Di Caro, R. Doursat, A. Engelbrecht, D. Floreano, L. Gambardella, R. Groß, E. Sahin, H. Sayama, and T. Stützle, editors, *Swarm Intelligence*, volume 6234 of *Lecture Notes in Computer Science*, pages 440–447. Springer, 2010.

[142] S. Konur, C. Dixon, and M. Fisher. Analysing robot swarm behaviour via probabilistic model checking. *Robotics and Autonomous Systems*, 60(2):199–213, 2012.

[143] P. Kouvaros and A. Lomuscio. Automatic verification of parameterised multi-agent systems. In *Proceedings of the 12th International Conference on Autonomous Agents and Multi-agent Systems (AAMAS'13)*, pages 861–868, Richland, SC, 2013.

[144] P. Kouvaros and A. Lomuscio. A cutoff technique for the verification of parameterised interpreted systems with parameterised environments. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI'13)*, pages 2013–2019. AAAI Press, 2013.

[145] R. Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 2(4):255–299, 1990.

[146] M. J. B. Krieger and J.-B. Billeter. The call of duty: Self-organised task allocation in a population of up to twelve mobile robots. *Robotics and Autonomous Systems*, 30(1):65–84, 2000.

[147] M. Kwiatkowska, A. Lomuscio, and H. Qu. Parallel model checking for temporal epistemic logic. In *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI'10)*, pages 543–548, Amsterdam, The Netherlands, 2010. IOS Press.

[148] M. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic symbolic model checker. In T. Field, P. Harrison, J. Bradley, and U. Harder, editors, *Computer Performance Evaluation: Modelling Techniques and Tools*, volume 2324 of *Lecture Notes in Computer Science*, pages 113–140. Springer, 2002.

[149] M. Kwiatkowska, G. Norman, D. Parker, and H. Qu. Assume-guarantee verification for probabilistic systems. In J. Esparza and R. Majumdar, editors, *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'10)*, volume 6105 of *Lecture Notes in Computer Science*, pages 23–37. Springer, 2010.

[150] J. E. Laird, A. Newell, and P. S. Rosenbloom. SOAR: an architecture for general intelligence. *Artificial Intelligence*, 33(1):1–64, Sept. 1987.

[151] I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan. Runtime assurance based on formal specifications. Departmental paper, Department of Computer & Information Science, University of Pennsylvania, 1999.

[152] L. Leemis. Simulation input modeling. In *Proceedings of the 31st Winter Simulation Conference (WSC'99)*, pages 14–23, 1999.

[153] A. Legay, B. Delahaye, and S. Bensalem. Statistical model checking: an overview. In H. Barringer, Y. Falcone, G. Roşu, B. Finkbeiner, O. Sokolsky, K. Havelund, I. Lee, G. Pace, and N. Tillmann, editors, *Proceedings of the 1st International Conference on Runtime Verification (RV'10)*, pages 122–135. Springer, 2010.

[154] R. Leombruni and M. Richiardi. Why are economists sceptical about agent-based simulations? *Physica A: statistical Mechanics and Its Applications*, 355:103–109, 2005.

[155] M. Leucker and C. Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293 – 303, 2009.

[156] D. K. Lewis. *On the Plurality of Worlds*. Blackwell Publishers, 1986.

[157] J. Li and U. Wilensky. NetLogo Sugarscape 1 immediate growback model. Technical report, Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL, 2009.

[158] X. Li, W. Mao, D. Zeng, and F.-Y. Wang. Agent-based social simulation and modeling in social computing. In C. Yang, H. Chen, M. Chau, K. Chang, S.-D. Lang, P. Chen, R. Hsieh, D. Zeng, F.-Y. Wang, K. Carley, W. Mao, and J. Zhan, editors, *Intelligence and Security Informatics*, volume 5075 of *Lecture Notes in Computer Science*, pages 401–412. Springer, 2008.

[159] W. Liu, A. Winfield, and J. Sa. Modelling swarm robotic systems: A case study in collective foraging. In M. S. Wilson, F. Labrosse, U. Nehmzow, C. Melhuish, and M. Witkowski, editors, *Towards Autonomous Robotic Systems*, pages 25–32, 2007.

[160] W. Liu, A. Winfield, J. Sa, J. Chen, and L. Dou. Strategies for energy optimisation in a swarm of foraging robots. In E. Şahin, W. Spears, and A. Winfield, editors, *Swarm Robotics*, volume 4433 of *Lecture Notes in Computer Science*, pages 14–26. Springer, 2007.

[161] A. Lomuscio, W. Penczek, and H. Qu. Partial order reductions for model checking temporal-epistemic logics over interleaved multi-agent systems. *Fundamenta Informaticae*, 101(1-2):71–90, Januar 2010.

[162] A. Lomuscio, W. Penczek, and B. Wożna. Bounded model checking for knowledge and real time. *Artificial Intelligence*, 171(16-17):1011 – 1038, 2007.

[163] A. Lomuscio, H. Qu, and F. Raimondi. MCMAS: A model checker for the verification of multi-agent systems. In A. Bouajjani and O. Maler, editors, *Computer Aided Verification*, volume 5643 of *Lecture Notes in Computer Science*, pages 682–688. Springer, 2009.

[164] A. Lomuscio and F. Raimondi. Model checking knowledge, strategies, and games in multi-agent systems. In *Proceedings of the 5th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS '06)*, pages 161–168, 2006.

[165] M. Luck and M. d'Inverno. A formal framework for agency and autonomy. In L. Gasser and V. Lesser, editors, *Proceedings of the 1st International Conference on Multi-Agent Systems*, pages 254–260. AAAI Press/MIT Press, 1995.

[166] S. Luke, C. Cioffi-Revilla, L. Panait, K. Sullivan, and G. Balan. MASON: a multi-agent simulation environment. *Transactions of the Society for Modeling and Simulation International*, 82(7):517–527, 2005.

[167] C. Macal and M. North. Tutorial on agent-based modeling and simulation part 2: How to model with agents. In *Proceedings of the 38th Winter Simulation Conference (WSC '05)*, pages 73–83, 2006.

[168] C. M. Macal and M. J. North. Tutorial on agent-based modeling and simulation. In *Proceedings of the 37th Winter Simulation Conference (WSC '05)*, pages 2–15, 2005.

[169] C. M. Macal and M. J. North. Agent-based modeling and simulation: desktop ABMs. In *Proceedings of the 39th Winter Simulation Conference (WSC '07)*, pages 95–106, 2007.

[170] M. W. Macy and R. Willer. From factors to actors: Computational sociology and agent-based modeling. *Annual Review of Sociology*, 28(1):143–166, 2002.

[171] B. Mahony and J. S. Dong. Timed Communicating Object Z. *IEEE Transactions on Software Engineering*, 26(2):150 –177, February 2000.

[172] B. P. Mahony and J. S. Dong. Blending Object-Z and Timed CSP: an introduction to TCOZ. In *Proceedings of the 20th International Conference on Software Engineering (ICSE'98)*, pages 95–104, Washington, DC, USA, 1998. IEEE Computer Society.

[173] Z. Manna and A. Pnueli. *Temporal verification of reactive systems: safety*. Springer, New York, NY, USA, 1995.

[174] C. Marchionni and P. Ylikoski. Generative Explanation and Individualism in Agent-Based Simulation. *Philosophy of the Social Sciences*, July 2013.

[175] E. Mares. Relevance logic. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Summer 2012 edition, 2012.

[176] N. Markey. Temporal logic with past is exponentially more succinct. *EATCS Bulletin*, 79:122–128, February 2003.

[177] R. E. Marks. Validating simulation models: A general framework and four applied examples. *Computational Economics*, 30:265–290, October 2007.

[178] P. McBurney. What are models for? In M. Cossentino, M. Kaisers, K. Tuyls, and G. Weiss, editors, *Multi-Agent Systems*, volume 7541 of *Lecture Notes in Computer Science*, pages 175–188. Springer, 2012.

[179] R. McCune and G. Madey. Agent-based simulation of cooperative hunting with UAVs. In *Proceedings of the Agent-Directed Simulation Symposium*. Society for Computer Simulation International, 2013.

[180] B. McKelvey. *The Blackwell Companion to Organizations*, chapter Model-centered organization science epistemology, pages 752–780. Blackwell, 2002.

[181] K. L. McMillan. *Symbolic model checking: an approach to the state explosion problem*. PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, USA, 1992.

[182] P. Menzies. Counterfactual theories of causation. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Spring 2014 edition, 2014.

[183] T. Menzies and C. Pecheur. Verification and validation and artificial intelligence. volume 65 of *Advances in Computers*, pages 153 – 201. Elsevier, 2005.

[184] B. Meyer. Applying "Design by Contract". *IEEE Computer*, 25(10):40–51, 1992.

[185] B. Meyer and C. Baudoin. *Méthodes de programmation*. Editions Eyrolles, third edition, 1984.

[186] D. Midgley, R. E. Marks, and D. Kunchamwar. Building and assurance of agent-based models: An example and challenge to the field. *Journal of Business Research*, 60(8):884 – 893, 2007. Special Issue on Complexities in Markets.

[187] J. H. Miller and S. E. Page. *Complex Adaptive Systems: An Introduction to Computational Models of Social Life*. Princeton Studies in Complexity. Princeton University Press, Princeton, NJ, USA, 2007.

[188] T. Miller and P. McBurney. Multi-agent system specification using TCOZ. In T. Eymann, F. Klügl, W. Lamersdorf, M. Klusch, and M. Huhns, editors, *Multiagent System Technologies*, volume 3550 of *Lecture Notes in Computer Science*, pages 216–221. Springer, 2005.

[189] R. Milner. *Communication and Concurrency*. Prentice-Hall, Upper Saddle River, NJ, USA, 1989.

[190] C. Ming and H. Schlingloff. Online monitoring of distributed systems with a five valued LTL. In *IEEE 44th International Symposium on Multiple-Valued Logic*, 2014.

[191] M. Mitzenmacher and E. Upfal. *Probability and Computing - Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[192] A. Morgenstern, M. Gesell, and K. Schneider. An asymptotically correct finite path semantics for LTL. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 304–319. Springer, 2012.

[193] J. Mount. A bit more on sample size. http://www.win-vector.com/blog/2013/03/a-bit-more-on-sample-size/, March 2013. Last accessed: 08/2014.

[194] M. Niazi, A. Hussain, and M. Kolberg. Verification and validation of agent based simulations using the VOMAS approach. In *Proceedings of the 3rd Workshop on Multi-Agent Systems and Simulation (MASS '09)*, 2009.

[195] S. E. Page. Aggregation in agent-based models of economies. *Knowledge Engineering Review*, 27:151–162, 2012.

[196] H. V. D. Parunak, R. Savit, and R. L. Riolo. Agent-based modeling vs. equation-based modeling: A case study and users' guide. In *Proceedings of the 1st International Workshop on Multiagent Systems and Agent-Based Simulation (MABS'98)*, pages 10–25, London, UK, 1998. Springer.

[197] J. Pearl. Probabilities of causation: three counterfactual interpretations and their identification. *Synthese*, 121:93–149, 1999.

[198] J. Pearl. *Causality: Models, Reasoning and Inference*. Cambridge University Press, 2000.

[199] D. Phan and F. Varenne. Agent-based models and simulations in economics and social sciences: From conceptual exploration to distinct ways of experimenting. *Journal of Artificial Societies and Social Simulation*, 13(1):5, 2010.

[200] S. Phelps, P. McBurney, and S. Parsons. A novel method for strategy acquisition and its application to a double-auction market game. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, 40(3):668–674, 2010.

[201] A. Pnueli and A. Zaks. PSL model checking and run-time verification via testers. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *Proceedings of the 14th International Conference on Formal Methods (FM'06)*, pages 573–586. Springer, 2006.

[202] A. Pnueli and A. Zaks. On the merits of temporal testers. In O. Grumberg and H. Veith, editors, *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*, pages 172–195. Springer, 2008.

[203] K. Preston White and R. G. Ingalls. Introduction to simulation. In *Proceedings of the 41st Winter Simulation Conference (WSC'09)*, pages 12–23, 2009.

[204] J.-F. Raskin. *Logics, automata and classical theories for deciding real time*. PhD thesis, University of Namur, 1999.

[205] S. Raychaudhuri. Introduction to Monte Carlo simulation. In *Proceedings of the 40th Winter Simulation Conference (WSC '08)*, pages 91–100, 2008.

[206] A. Regayeg, A. Hadj Kacem, and M. Jmaiel. Specification and verification of multi-agent applications using Temporal Z. In *Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT 2004)*, pages 260 – 266, 2004.

[207] C. W. Reynolds. Flocks, herds and schools: A distributed behavioral model. *SIGGRAPH Computer Graphics*, 21:25–34, August 1987.

[208] B. D. Ripley. Selecting amongst large classes of models. *Methods and models in statistics: In honor of Professor John Nelder, FRS*, pages 155–170, 2004.

[209] G. Roşu, W. Schulte, and T. Şerbănuţă. Runtime verification of C memory safety. In S. Bensalem and D. Peled, editors, *Runtime Verification*, volume 5779 of *Lecture Notes in Computer Science*, pages 132–151. Springer, 2009.

[210] A. Roscoe. *Understanding Concurrent Systems*. Springer, London, UK, 2010.

[211] A. W. Roscoe, C. A. R. Hoare, and R. Bird. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[212] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, second edition, 2003.

[213] P. Salem da Silva and A. De Melo. An approach for the verification of multi-agent systems by formally guided simulations. In *Proceedings of the International Joint Conferences on Web Intelligence and Intelligent Agent Technologies (WI-IAT '13)*, volume 2, pages 266–273, 2013.

[214] R. G. Sargent. Statistical analysis of simulation output data. *SIGSIM Simulation Digest*, 8:21–31, April 1977.

[215] R. G. Sargent. Verifying and validating simulation models. In *Proceedings of the 28th Winter Simulation Conference (WSC'96)*, pages 55–64, 1996.

[216] R. G. Sargent. Verification and validation of simulation models. In *Proceedings of the 40th Winter Simulation Conference (WSC '08)*, pages 157–169, 2008.

[217] J. Schaffer. The metaphysics of causation. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Fall 2008 edition, 2008.

[218] T. C. Schelling. Models of segregation. *American Economic Review*, 59(2):488–93, May 1969.

[219] A. Schmid. What is the truth of simulation? *Journal of Artificial Societies and Social Simulation*, 8(4):5, 2005.

[220] J. R. Searle. *Speech Acts: An Essay in the Philosophy of Language*. Cambridge University Press, 1969.

[221] R. E. Shannon. Introduction to the art and science of simulation. In *Proceedings of the 30th Winter Simulation Conference (WSC'98)*, pages 7–14, 1998.

[222] J. Shore. Fail fast. *IEEE Software*, 21(5):21–25, Sept. 2004.

[223] I. Shpitser and J. Pearl. What counterfactuals can be tested. *Proceedings of the 23rd Conference on Uncertainty in Artificial Intelligence*, pages 352–359, 2007.

[224] G. Smith. A fully abstract semantics of classes for Object-Z. *Formal Aspects of Computing*, 7(3):289–313, 1995.

[225] G. Smith. A semantic integration of Object-Z and CSP for the specification of concurrent systems. In J. S. Fitzgerald, C. B. Jones, and P. Lucas, editors, *Proceedings of the 4th International Symposium of Formal Methods Europe on Industrial Applications and Strengthened Foundations of Formal Methods (FME'97)*, pages 62–81, London, UK, 1997. Springer.

[226] G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[227] G. Smith and J. Derrick. Refinement and verification of concurrent systems specified in Object-Z and CSP. In M. Hinchey and S. Liu, editors, *Proceedings of the 1st International Conference on Formal Engineering Methods (ICFEM'97)*, pages 182–196, 1997.

[228] J. M. Spivey. *The Z notation: A Reference Manual*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1992.

[229] P. Suppes. *A Probabilistic Theory of Causality*. North-Holland Publishing Company, Amsterdam, 1970.

[230] R. Temam. *Navier-Stokes Equations: Theory and Numerical Analysis*. American Mathematical Society, 2000.

[231] W. Van Der Hoek and M. Wooldridge. Tractable multiagent planning for epistemic goals. In M. Gini, T. Ishida, C. Castelfranchi, and W. L. Johnson, editors, *Proceedings of the 1st International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'02)*, pages 1167–1174, 2002.

[232] M. Vardi. Automata-theoretic model checking revisited. In B. Cook and A. Podelski, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 4349 of *Lecture Notes in Computer Science*, pages 137–150. Springer, 2007.

[233] R. Vaughan. Massively multi-robot simulation in stage. *Swarm Intelligence*, 2(2-4):189–208, 2008.

[234] A. Waldherr and N. Wijermans. Communicating social simulation models to sceptical minds. *Journal of Artificial Societies and Social Simulation*, 16(4):13, 2013.

[235] W. Wan, J. Bentahar, and A. Ben Hamza. Model checking epistemic and probabilistic properties of multi-agent systems. In *Modern Approaches in Applied Intelligence*, volume 6704 of *Lecture Notes in Computer Science*, pages 68–78. Springer, 2011.

[236] Y. Wei, G. Madey, and M. Blake. Agent-based simulation for UAV swarm mission planning and execution. In *Proceedings of the Agent-Directed Simulation Symposium*, page 2. Society for Computer Simulation International, 2013.

[237] J. W. Weibull. *Evolutionary game theory*. MIT press, 1997.

[238] G. Weiss. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. MIT Press, 1999.

[239] D. Weyns, H. Parunak, F. Michel, T. Holvoet, and J. Ferber. Environments for multiagent systems: state-of-the-art and research challenges. In *Proceedings of the 1st International Conference on Environments for Multi-Agent Systems (E4MAS'04)*, pages 1–47. Springer, 2005.

[240] U. Wilensky. NetLogo. Technical report, Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL., 1999.

[241] P. Windrum, G. Fagiolo, and A. Moneta. Empirical validation of agent-based models: Alternatives and prospects. *Journal of Artificial Societies and Social Simulation*, 10(2), 2007.

[242] J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice-Hall, Inc., 1996.

[243] M. J. Wooldridge. *Introduction to Multiagent Systems*. John Wiley & Sons, Inc., New York, NY, USA, 2001.

[244] M. J. Wooldridge, M. Fisher, M.-P. Huget, and S. Parsons. Model checking multi-agent systems with MABLE. In M. Gini, T. Ishida, C. Castelfranchi, and W. L. Johnson, editors, *Proceedings of the 1st International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'02)*, pages 952–959, 2002.

[245] M. J. Wooldridge and N. R. Jennings. Intelligent agents: theory and practice. *The Knowledge Engineering Review*, 10(02):115–152, 1995.

[246] C. J. Wright, P. McMinn, and J. Gallardo. Towards the automatic identification of faulty multi-agent based simulation runs using MASTER. In F. Giardini and F. Amblard, editors, *Multi-Agent-Based Simulation XIII*, volume 7838 of *Lecture Notes in Computer Science*, pages 143–156. Springer, 2013.

[247] B. Zhang, W. K. Chan, and S. Ukkusuri. Agent-based discrete-event hybrid space modeling approach for transportation evacuation simulation. In *Proceedings of the 2011 Winter Simulation Conference*, pages 199–209, 2011.